UNIT - IV

Introduction to Structures

A **structure** is a **user-defined data type** that allows the combination of **different data types** into a single logical unit. This helps represent a record or an entity having various attributes of possibly different types.

For example, a student record may have:

- A name (string)
- Roll number (integer)
- Marks (float)

A structure groups all of these under a single name.

Definition of a Structure

The **struct keyword** is used to define a structure.

It serves as a blueprint or template, but **does not allocate memory** until a structure variable is declared.
The items or fields declared inside a structure are called its **members**.
Each member can be of **any valid C data type** (e.g., int, float, char, arrays, or even other structures).

Syntax:

```
struct structure_name {
   data_type member1;
   data_type member2;
   ...
};
```

Here:

- struct → Keyword used to define a structure.
- structure_name → Name given to the structure type.
- member1, member2, etc. → Data items or variables grouped under that structure.

Example:

```
struct Student {
   int roll_no;
```

```
    char name[30];
    float marks;
};
```

This defines a structure named Student containing:

- An integer variable roll_no, a character array name (string), and a float variable marks.

Declaring Structure Variables

Declaring a structure variable is what actually **allocates memory** for the structure's members. There are three common ways to declare structure variables:

1. Separate Declaration

2. Declaration with Definition

3. Declaration using typedef

1. Separate Declaration

A Separate Declaration is when the structure is defined first using the struct keyword and the variables are declared later in the program.

It allows the same structure type to be reused anywhere in the program to create multiple structure variables.

Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
};
struct structure_name variable1, variable2;
```

Eaxmple:

```
struct Student {
    int roll_no;
```

```
    char name[30];
    float marks;
};
struct Student s1, s2;   // Structure variables
```

Advantages:

- The structure type can be reused anywhere in the program.
- Easy to declare more variables later as needed.
- Good for large programs or when the structure is used in multiple files/functions.

Disadvantages:

- You must use the struct keyword every time (struct Student s1;).
- Slightly more verbose in declaration.

2. Declaration with Definition

In Declaration with Definition**,** the structure **is** defined and its variables are declared at the same time**.**

This method is compact and useful when the structure is required only in a specific part of the program.

Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
} variable1, variable2;
```

Example:

```
struct Employee {
    int emp_id;
    char emp_name[20];
    float salary;
} e1, e2;
```

Advantages:

- Compact and simple — structure definition and variable declaration done together.
- Saves lines of code for small or single-use programs**.**

Disadvantages:

- The structure name is defined locally here; you can't reuse it easily to declare new variables later (unless you repeat the definition).
- Not suitable for large programs where the structure needs to be shared.

3. Declaration using typedef

The typedef declaration creates a user-defined alias name for a structure type, allowing variables to be declared without using the struct keyword.

It makes the code shorter, cleaner, and easier to read, especially in large programs.

Syntax:

```
typedef struct {
   data_type member1;
   data_type member2;
   ...
} alias_name;
alias_name variable1, variable2;
```

Example:

```
typedef struct {
   int book_id;
   char title[30];
   float price;
} Book;
```

Book b1, b2;

Advantages:

- Removes the need to repeatedly write struct.
- Makes the code cleaner, shorter, and more readable.
- Commonly used in professional coding and function parameters**.**

Disadvantages:

- Slightly less explicit**,** since the struct keyword is hidden — can confuse beginners.
- Type aliasing may hide the real structure tag name (less clarity in debugging).

Accessing Structure Members

The two main operators for accessing structure members are the dot operator (.) and the arrow operator **(->)**.

1. The Dot Operator (.)

The dot operator (or member access operator) is used when you have the actual structure variable itself. It connects the structure variable name with its member name.

Syntax:

structure_variable.member_name;

Example:

```
#include <stdio.h>

struct Student {

   int id;

   char name[20];

};

int main() {

   struct Student s1 = {101, "Alice"};

   printf("ID: %d\n", s1.id);

   printf("Name: %s\n", s1.name);
```

```
    return 0;
}
```

Output:

ID: 101

Name: Alice

Advantages

1. Simple and easy to use when the structure variable is directly available.
2. No need for pointers — reduces complexity.
3. Safe, since no pointer dereferencing is involved.
4. Good for **small** and local structures**.**
5. Readability is high — clean syntax.

Disadvantages

1. Cannot be used with structure pointers**.**
2. Makes a copy of structure members if passed to a function — not memory efficient.
3. Accessing nested or large data through copies can reduce performance.
4. Not suitable when you want to modify original structure da

When to Use

- When you have a simple structure variable (not a pointer).
- When no modification to original data is required.
- When dealing with local or small data structures.

When Not to Use

- When the structure is large (copying data is costly).
- When you need to pass the structure to a function by reference.
- When working with dynamically allocated structures.

2. The Arrow Operator (->)

The arrow operator (or structure pointer operator) is used when you have a pointer that stores the address of a structure variable. It combines dereferencing (*) and member access (.) in a single operator.

Syntax:

pointer_to_structure->member_name;

Example:

```
#include <stdio.h>

struct Student {

    int id;

    char name[20];

};

int main() {

    struct Student s1 = {102, "Bob"};

    struct Student *ptr = &s1;

    printf("ID: %d\n", ptr->id);

    printf("Name: %s\n", ptr->name);

    return 0;

}
```

Output:

ID: 102

Name: Bob

Advantages

1. Efficient for large structures — avoids copying entire data.
2. Allows direct access to the original structure data via pointer.
3. Essential for dynamic memory allocation (malloc, calloc).
4. Easier to pass structures to functions by reference.
5. Supports linked lists, trees, and other data structures.

Disadvantages

1. Requires pointers — more complex and error-prone.
2. Risk of invalid memory access (dangling pointers).
3. Needs proper memory management (allocation & deallocation).
4. Harder to debug compared to dot operator.
5. Requires -> instead of . — may confuse beginners.

When to Use

- When you are using structure pointers.
- When structure data is allocated dynamically**.**
- When you want to modify the original structure data inside a function.
- When implementing linked lists, trees, or graphs**.**

When Not to Use

- When no pointer is needed (simple data access).
- When structure is small and local**.**
- When you are not familiar with pointer handling and memory management**.**

Array of Structures in C

An Array of Structures is a collection of structure variables that share the same data type.
It allows storing multiple records of the same kind (e.g., students, employees, products) in a single structure array**.**

It combines the power of arrays (sequential data storage) and structures (grouping of different data types).

Syntax:

```
struct structure_name {

   data_type member1;

   data_type member2;

   ...

};

struct structure_name array_name[size];
```

Example:

```
#include <stdio.h>
struct Student {
    int id;
    char name[20];
    float marks;
};
int main() {
    // Declare array of structure
    struct Student s[3] = {
        {1, "Alice", 85.5},
        {2, "Bob", 90.0},
        {3, "Charlie", 78.5}
    };
    printf("Student Details:\n");
    for(int i = 0; i < 3; i++) {
        printf("ID: %d\n", s[i].id);
        printf("Name: %s\n", s[i].name);
        printf("Marks: %.2f\n\n", s[i].marks);
    }
    return 0;
}
```

Output:

Student Details:

ID: 1

Name: Alice

Marks: 85.50

ID: 2

Name: Bob

Marks: 90.00

ID: 3

Name: Charlie

Marks: 78.50

Nested Structure in C (Structure within Structure)

A Nested Structure means defining one structure inside another structure**.**
It allows grouping related but complex data logically by creating hierarchical relationships between structures.

In simple terms, **a** member of one structure is itself another structure**.**

Syntax:

```
struct outer_structure {
    data_type member1;
    struct inner_structure {
        data_type sub_member1;
```

```
        data_type sub_member2;
    } inner_var; // inner structure variable
    data_type member2;
};
```

Or you can define the inner structure separately**:**

```
struct inner_structure {
    data_type sub_member1;
    data_type sub_member2;
};
struct outer_structure {
    data_type member1;
    struct inner_structure inner_var;
    data_type member2;
};
```

Example:

```
#include <stdio.h>
struct Address {
    char city[20];
    int pincode;
};
struct Student {
    int id;
```

```
    char name[20];
    struct Address addr;  // Nested structure
};
int main() {
    struct Student s1 = {101, "Alice", {"Chennai", 600001}};
    printf("Student ID: %d\n", s1.id);
    printf("Name: %s\n", s1.name);
    printf("City: %s\n", s1.addr.city);
    printf("Pincode: %d\n", s1.addr.pincode);
    return 0;
}
```

Output:

Student ID: 101

Name: Alice

City: Chennai

Pincode: 600001

Structure as Function Parameter

In C, a structure **c**an be passed to a function just like any other variable.
This allows functions to operate on structured data such as records (students, employees, etc.).

There are two main ways to pass a structure to a function:

1. **Pass by Value** – Passing a copy of the structure.
2. **Pass by Reference** – Passing the address (pointer) of the structure.

1. Pass by Value

In Pass by Value, a copy of the structure is passed to the function.
The original structure remains unchanged, even if the function modifies the copy.

Syntax:

return_type function_name(struct structure_name variable);

Function Call:

function_name(struct_variable);

Example:

```
#include <stdio.h>

struct Student {

    int id;

    float marks;

};

void display(struct Student s) {

    s.marks = 95.0;  // modifies local copy only

    printf("Inside Function: ID = %d, Marks = %.2f\n", s.id, s.marks);

}

int main() {

    struct Student s1 = {101, 85.0};

    display(s1);  // pass by value

    printf("In Main: ID = %d, Marks = %.2f\n", s1.id, s1.marks);

    return 0;
```

```
}
```

Output:

Inside Function: ID = 101, Marks = 95.00

In Main: ID = 101, Marks = 85.00

2. Pass by Reference

In Pass by Reference**,** the address of the structure (pointer) is passed to the function.
The function can directly modify the original structure data.

Syntax:

```
return_type function_name(struct structure_name *variable);
```

Function Call:

```
function_name(&struct_variable);
```

Example:

```
#include <stdio.h>

struct Student {
    int id;
    float marks;
};

void update(struct Student *s) {
    s->marks = 95.0;  // modifies original structure
    printf("Inside Function: ID = %d, Marks = %.2f\n", s->id, s->marks);
}

int main() {
```

```
    struct Student s1 = {102, 85.0};

    update(&s1);  // pass by reference

    printf("In Main: ID = %d, Marks = %.2f\n", s1.id, s1.marks);

    return 0;

}
```

Output:

Inside Function: ID = 102, Marks = 95.00

In Main: ID = 102, Marks = 95.00

Structure Returning from Function

A structure returning from a function in C refers to the process where a function returns an entire structure variable as its result instead of a single value.
This allows the function to send multiple related data items (grouped under a structure) back to the calling function in a single return statement.

Syntax:

struct structure_name function_name(parameters);

Returning a structure:

return structure_variable;

Function call:

struct structure_name variable = function_name(arguments);

Example:

```
#include <stdio.h>

struct Student {

    int id;
```

```
    float marks;
};
// Function returning structure
struct Student getStudentData() {
    struct Student s;
    printf("Enter ID and Marks: ");
    scanf("%d %f", &s.id, &s.marks);
    return s;   // returns structure variable
}
int main() {
    struct Student s1;
    s1 = getStudentData();  // function returns structure
    printf("\nStudent Details:\n");
    printf("ID = %d\n", s1.id);
    printf("Marks = %.2f\n", s1.marks);

    return 0;
}
```

Output:

```
Enter ID and Marks: 101 88.5
Student Details:
ID = 101
```

Marks = 88.50

typedef Keyword

The typedef keyword in C is used to create an alias (alternative name) for an existing data type.
It helps make complex declarations easier to read, improves program clarity, and provides flexibility for future modifications.

One of the most common uses of typedef is with structures**,**
because it removes the need to repeatedly write the keyword struct.

Syntax:

typedef existing_data_type new_name;

Example:

```
typedef unsigned int uint;

int main() {

    uint age = 25;   // same as: unsigned int age = 25;

    printf("%u", age);

    return 0;

}
```

Output:

25

Syntax:

```
typedef struct structure_name {

    data_type member1;

    data_type member2;

} alias_name;
```

Example with Structures:

```
#include <stdio.h>

typedef struct {
    int id;
    char name[20];
    float marks;
} Student;  // alias name for the structure

int main() {
    Student s1 = {101, "Alice", 88.5};
    printf("ID: %d\n", s1.id);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);
    return 0;
}
```

Output:

ID: 101

Name: Alice

Marks: 88.50

UNION

A Union is a user-defined data type like a structure, but a union can store only one value at a time, even though it may contain multiple members of different data types.

The memory allocated to a union is equal to the size of its largest member**.**
Unions are mainly used for efficient memory management and data interpretation.

Syntax:

```
union union_name {
    data_type member1;
    data_type member2;
    ...
};
```

Example:

```
#include <stdio.h>
union Data {
    int i;
    float f;
    char ch;
};
int main() {
    union Data d1;
    d1.i = 10;
    printf("Integer: %d\n", d1.i);
    d1.f = 20.5;
    printf("Float: %.2f\n", d1.f);
    d1.ch = 'A';
    printf("Character: %c\n", d1.ch);
    // Printing all members
```

```
    printf("\nAfter assigning char:\n");
    printf("Integer: %d\n", d1.i);
    printf("Float: %.2f\n", d1.f);
    printf("Character: %c\n", d1.ch);
    return 0;
}
```

Output:

Integer: 10

Float: 20.50

Character: A

After assigning char:

Integer: 1094795585

Float: 0.000000

Character: A

Advantages of Union:

- **Memory Efficient** – Uses the same memory for all members.
- **Useful for Embedded Systems** – Helps handle multiple data formats efficiently.
- **Simplifies Program Logic** – Ideal when a variable may hold different data types at different times.
- **Can Store Different Data Types** – Flexible data handling under one name.
- **Faster Access** – Accessing members is direct, similar to structures.

Disadvantages of Union:

- **Stores Only One Value at a Time** – Other members' values get overwritten.
- **Data Loss Risk** – Assigning a new member destroys the previous one.

- **Difficult Debugging** – Hard to track which member is currently active.
- **Not Suitable for Complex Data Models** – Use structures for that.
- **No Type Safety** – Programmer must ensure the correct member is accessed.

Declaring and Initializing Union Variables

There are three main ways to declare and initialize unions:

(a) Separate Declaration

```
union Student s1;
```

(b) Declaration with Definition

```
union Student {
    int id;
    char name[20];
    float marks;
} s1;
```

(c) Using typedef

```
typedef union {
    int id;
    char name[20];
    float marks;
} Student;
Student s1;
```

Accessing Union Members

Union members can be accessed using the dot operator (.) for a union variable or the arrow operator (->) for a union pointer.

Example:

```
union Sample {
    int id;
    float marks;
};
int main() {
    union Sample s1, *ptr;
    ptr = &s1;
    s1.id = 101;
    printf("Using dot: %d\n", s1.id);
    ptr->marks = 88.5;
    printf("Using arrow: %.2f\n", ptr->marks);
}
```

Output:

Using dot: 101

Using arrow: 88.50

Nested Union (Union within Union)

A Nested Union (also called Union within Union) is a type of union in which one union is declared as a member of another union**.** In this type, an inner union becomes part of an outer union**,** allowing multiple groups of related data to share the same memory space.

A Nested Union lets you store different sets of data, where each set can contain multiple data types, but still shares one memory block at a time.

Nested unions are mainly used when:

1. You want to group related data in separate unions (inner ones), but still save memory by sharing storage.
2. Only one set of data (inner union or outer member) is needed at a time.
3. To represent multi-format or multi-type records, such as:
    - Sensor data with multiple modes
    - Network packets with varying content types
    - Hardware register configurations

Syntax:

```
union OuterUnion {

   data_type member1;

   union InnerUnion {

      data_type member2;

      data_type member3;

   } inner_var;

};
```

Example:

```
#include <stdio.h>

#include <string.h>

union Inner {

   float price;

   char name[20];

};
```

```
union Outer {
    int id;
    union Inner item;  // Nested Union
};
int main() {
    union Outer product;
    product.id = 101;
    printf("Product ID: %d\n", product.id);
    product.item.price = 99.50;
    printf("Product Price: %.2f\n", product.item.price);
    strcpy(product.item.name, "Notebook");
    printf("Product Name: %s\n", product.item.name);
    return 0;
}
```

Output:

Product ID: 101

Product Price: 99.50

Product Name: Notebook

Union within Structure

A Union within a Structure means that a union is declared as a member of a structure**.**

It allows combining the advantages of both structures and unions:

- Structure helps store multiple fields of different types together.

- Union helps share memory among certain members that are mutually exclusive**.**

Union within structure is useful when:

1. You need to store common data (like IDs, names, etc.)
   along with variable data (like different types of values) in the same record.
2. You want memory-efficient structures when not all fields are used at the same time.
3. You handle records or data packets that can have different formats **or** types.

Syntax:

```
struct StructureName {
    data_type member1;
    data_type member2;
    union UnionName {
        data_type member3;
        data_type member4;
    } union_variable;
};
```

Example:

```
#include <stdio.h>
#include <string.h>
struct Sensor {
    char type[10];
    union Data {
        int temperature;
        float pressure;
```

```
        char status[10];
    } data;
};
int main() {
    struct Sensor s;
    strcpy(s.type, "Temp");
    s.data.temperature = 36;
    printf("%s Sensor Reading: %d°C\n", s.type, s.data.temperature);
    strcpy(s.type, "Pressure");
    s.data.pressure = 101.3;
    printf("%s Sensor Reading: %.2f kPa\n", s.type, s.data.pressure);
    strcpy(s.type, "Status");
    strcpy(s.data.status, "Active");
    printf("%s Sensor Status: %s\n", s.type, s.data.status);
    return 0;
}
```

Output:

Temp Sensor Reading: 36°C

Pressure Sensor Reading: 101.30 kPa

Status Sensor Status: Active

Structure within Union

A Structure within a Union means that a **structure** is declared as a member inside a union**.**

This allows multiple sets of related data (structures) to share the same memory location, where only one structure (or another union member) can hold valid data at any given time.

A structure within a union is used when:

1. You have different record formats but only one format is active at a time.
2. You want to save memory while managing structured data groups.
3. You need a flexible data type that can represent multiple record layouts efficiently.

Common real-life uses:

- File systems (storing data of different file types)
- Network packets (header formats differ based on type)
- Embedded systems (registers with different field layouts)

Syntax:

```
union UnionName {
   data_type member1;
   struct StructureName {
      data_type field1;
      data_type field2;
   } struct_variable;
};
```

Example:

```
#include <stdio.h>
#include <string.h>
union Record {
   struct Student {
      char name[20];
```

```
        int roll;
    } s;
    struct Employee {
        char emp_name[20];
        float salary;
    } e;
};
int main() {
    union Record r;
    strcpy(r.s.name, "Amit");
    r.s.roll = 12;
    printf("Student Name: %s\n", r.s.name);
    printf("Roll No: %d\n", r.s.roll);
    strcpy(r.e.emp_name, "Ravi");
    r.e.salary = 55000.75;
    printf("\nEmployee Name: %s\n", r.e.emp_name);
    printf("Salary: %.2f\n", r.e.salary);
    return 0;
}
```

Output:

Student Name: Amit

Roll No: 12

Employee Name: Ravi

Salary: 55000.75

Difference Between Structure and Union

Structure (struct)	Union (union)
Each member of a structure has its own memory location.	All members of a union share the same memory location.
The size of a structure is the sum of the sizes of all its members.	The size of a union is equal to the size of its largest member.
All members can store and access values at the same time.	Only one member can hold a value at a time.
Changing one member's value does not affect others.	Changing one member's value overwrites the previous member's value.
More memory is required because each member gets its own space.	Less memory is required because memory is shared.
Used when all data fields are required simultaneously.	Used when only one data field is needed at a time.
Allows initialization of all members during declaration.	Allows initialization of only the first member during declaration.
Safer to use — data of all members remain valid.	Risky — only the last stored data remains valid.
Commonly used for records and data grouping (e.g., student info).	Commonly used for memory-saving or type-variant data (e.g., hardware registers).
Example: struct Student { int id; char name[20]; float marks; };	Example: union Student { int id; char name[20]; float marks; };

Dynamic Memory Allocation (DMA)

Dynamic Memory Allocation (DMA**)** refers to the process of allocating memory at runtime rather than at compile time.
It allows a program to request memory as needed while it is running, making it flexible and efficient for variable-sized data structures.

In normal (static) memory allocation:

- The size of arrays or variables must be known before compilation.
- Memory is allocated automatically and **b** throughout execution.

In dynamic allocation:

- Memory is allocated manually at runtime using library functions.
- Memory can be increased, decreased, or released when no longer needed.

All DMA functions are defined in the stdlib.h header file.

#include <stdlib.h>

In C programming, **memory** can be allocated in two ways:

- **Static memory allocation** → during compile time
- **Dynamic memory allocation** → during runtime

malloc() → Memory Allocation (Uninitialized)

The **malloc()** function (Memory Allocation) in C dynamically allocates a **single block of memory** of a specified size (in bytes) during runtime and returns a **pointer** to the first byte of the allocated block.

The memory allocated by malloc() is **uninitialized**, meaning it contains **garbage values** until explicitly assigned.

- Memory allocated by malloc() comes from the **heap** (dynamic memory area).
- It **does not initialize** the memory block.
- The pointer returned by malloc() is of type void *, so it must be **typecasted** to the required data type.
- If memory allocation fails, malloc() returns **NULL**.

Syntax:

ptr = (cast_type*) malloc(size_in_bytes);

Here,

ptr Pointer variable to hold the base address of allocated memory
cast_type* Type to which the pointer is cast (e.g., int*, float*)
malloc() Function that allocates memory dynamically
size_in_bytes Total number of bytes to allocate

Formula to Calculate Memory Size:

ptr = (data_type*) malloc(number_of_elements * sizeof(data_type));

Here,

sizeof(data_type) → gives the number of bytes required for one element.

number_of_elements → number of elements to allocate.

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int *ptr;

    int n, i;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    // Allocating memory for n integers

    ptr = (int*) malloc(n * sizeof(int));

    // Check if memory is allocated successfully

    if (ptr == NULL) {

        printf("Memory not allocated.\n");
```

```
        return 1;
    }
    printf("Memory successfully allocated using malloc.\n");
    // Assigning values to allocated memory
    for (i = 0; i < n; i++) {
        ptr[i] = i + 1;
    }
    // Displaying the values
    printf("The elements of the array are: ");
    for (i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    // Free the allocated memory
    free(ptr);
    return 0;
}
```

Output:

Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1 2 3 4 5

Checking for Successful Allocation

Example:

```
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
```

// If not checked, accessing a NULL pointer causes a **segmentation fault** (runtime error).

Releasing Allocated Memory : Memory allocated with malloc() should always be released using free() once it is no longer required.

Example:

```
free(ptr);
ptr = NULL;   // Good practice to avoid dangling pointer
```

Advantages of malloc()

1. Allocates memory **dynamically during runtime**
2. Avoids wastage of unused memory
3. Enables creation of **flexible data structures**
4. Works well with realloc() and free()
5. Returns a **pointer** that can be used as an array

Disadvantages

1. Memory is **uninitialized** (contains garbage).
2. Must manually check for NULL.
3. Requires explicit deallocation using free().
4. Can lead to **memory leaks** if not managed properly.
5. Slight overhead compared to static memory allocation.

calloc() → Contiguous Allocation (Initialized)

The calloc() function (Contiguous Allocation) in C dynamically allocates memory for an array of elements, each of a specified size, and initializes all bytes in the allocated storage to zero**.**

It returns a void pointer to the first byte of the allocated block, or NULL if the allocation fails.

- Used to allocate multiple memory blocks at once (e.g., arrays).

- Initializes all elements to zero (0**)** automatically.
- Allocates memory contiguously, ensuring elements are stored one after another.
- Returns a void * pointer which must be typecasted.
- Defined in the header file:

  #include <stdlib.h>

Syntax:

ptr = (cast_type*) calloc(num_of_elements, size_of_each_element);

Here,

ptr Pointer variable to the base address of the allocated memory block
cast_type* Type to which the pointer is cast (e.g., int*, float*, etc.)
num_of_elements Total number of elements to allocate
size_of_each_element Size (in bytes) of one element
calloc() Function that performs the allocation

Formula for memory size:

Total Memory = num_of_elements × size_of_each_element

This total number of bytes will be allocated **contiguously** and initialized to **zero**.

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

   int *ptr;

   int n, i;

   printf("Enter number of elements: ");

   scanf("%d", &n);
```

```
// Allocating memory using calloc
ptr = (int*) calloc(n, sizeof(int));
// Checking successful allocation
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    return 1;
}
printf("Memory successfully allocated using calloc.\n");
// Displaying initialized values
printf("The elements of the array are: ");
for (i = 0; i < n; i++) {
    printf("%d ", ptr[i]);
}
// Assigning new values
for (i = 0; i < n; i++) {
    ptr[i] = (i + 1) * 10;
}
// Displaying new values
printf("\nUpdated elements: ");
for (i = 0; i < n; i++) {
    printf("%d ", ptr[i]);
}
```

```
    free(ptr);

    return 0;

}
```

Output:

Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 0 0 0 0 0

Updated elements: 10 20 30 40 50

Advantages of calloc()

1. Automatically initializes all bytes to **zero**.
2. Allocates **contiguous memory blocks** for arrays.
3. Prevents **garbage values** in memory.
4. Reduces errors due to uninitialized memory access.
5. Useful when creating **multi-element data structures**.

Disadvantages

1. Slightly **slower than malloc()** (due to zero-initialization).
2. Must be **freed manually** using free().
3. Risk of **memory leaks** if not properly released.
4. If memory fails, returns NULL (must handle carefully).
5. Cannot allocate extremely large memory if heap is full.

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int *ptr, n, i, sum = 0;

    printf("Enter number of elements: ");
```

```
scanf("%d", &n);
// Memory allocation using calloc
ptr = (int*) calloc(n, sizeof(int));
// Check if memory is successfully allocated
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    return 1;
}
printf("Memory successfully allocated using calloc.\n");
// Input elements
printf("Enter %d elements:\n", n);
for (i = 0; i < n; i++) {
    scanf("%d", &ptr[i]);
}
// Display elements and calculate sum
printf("Array elements are: ");
for (i = 0; i < n; i++) {
    printf("%d ", ptr[i]);
    sum += ptr[i];
}
printf("\nSum of elements = %d\n", sum);
// Free allocated memory
```

```
    free(ptr);
    return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using calloc.
Enter 5 elements:
10
20
30
40
50
Array elements are: 10 20 30 40 50
Sum of elements = 150
```

realloc() → Re-allocation or Resize Memory

The **realloc()** (reallocation) function in C is used to **change the size of memory** that has already been dynamically allocated by **malloc()** or **calloc()**.
It can **increase or decrease** the size of the allocated block at runtime while **preserving the existing data** (up to the minimum of old and new sizes).

It is defined in the **<stdlib.h>** header file.

Syntax:

```
ptr = (cast_type*) realloc(ptr, new_size_in_bytes);
```

Here,

ptr Pointer to previously allocated memory block

new_size_in_bytes New size (in bytes) to be allocated

Return Value Returns a pointer to the new memory block, or NULL if reallocation fails

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int *ptr;

    int i;

    // Step 1: Allocate memory for 3 integers

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

    // Assign initial values

    for (i = 0; i < 3; i++)

        ptr[i] = i + 1;     // 1, 2, 3

    printf("Before reallocation:\n");

    for (i = 0; i < 3; i++)

        printf("%d ", ptr[i]);

    // Step 2: Reallocate memory for 5 integers

    ptr = (int*) realloc(ptr, 5 * sizeof(int));

    // Assign new values

    ptr[3] = 4;

    ptr[4] = 5;
```

```
    printf("\nAfter reallocation:\n");
    for (i = 0; i < 5; i++)
        printf("%d ", ptr[i]);
    free(ptr);
    return 0;
}
```

Output:

Before reallocation:

1 2 3

After reallocation:

1 2 3 4 5

free() → Deallocate or Release Memory

The **free()** function in C is used to **deallocate** or **release** memory that was previously allocated dynamically using malloc(), calloc(), or realloc().

When memory is no longer needed, freeing it helps **prevent memory leaks** and allows the operating system to **reuse that memory** for other processes.

It is defined in the **<stdlib.h>** header file.

Syntax:

free(pointer_name);

Here,

pointer_name
Pointer to the dynamically allocated memory block that you want to free

Note:

- Only memory allocated using malloc(), calloc(), or realloc() should be freed.
- After using free(), the pointer becomes a **dangling pointer** (i.e., points to invalid memory). To avoid this, **set it to NULL** after freeing.
- Freeing the same pointer twice causes **undefined behavior**.

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int *ptr;

    int i, n = 5;

    // Memory allocation

    ptr = (int*) malloc(n * sizeof(int));

    // Check allocation

    if (ptr == NULL) {

        printf("Memory not allocated.\n");

        return 1;

    }

    // Assign values

    for (i = 0; i < n; i++)

        ptr[i] = i + 1;

    printf("Array elements:\n");

    for (i = 0; i < n; i++)

        printf("%d ", ptr[i]);
```

```
    // Free memory
    free(ptr);
    ptr = NULL;  // Avoid dangling pointer
    printf("\n\nMemory successfully freed.\n");
    return 0;
}
```

Output:

Array elements:

1 2 3 4 5

Memory successfully freed.