UNIT - 4

Dr.Kavitha N
Associate Professor
Dept. of IT

UNIT- IV

Topics to be covered :

Structure, Union and Dynamic memory Allocation

Structure and Union Types: User-Defined Structure Types, Structure Type Data as Input and Output Parameters, Functions with Structured Result Values, Union Types.

Dynamic Memory Allocation: Introduction, malloc, calloc, realloc, free.

Dr.Kavitha N
Associate Professor
Dept. of IT

STRUCTURES IN C

A structure is a user-defined data type that allows combining variables of different data types into a single unit. It is mainly used to represent a record (like a student, employee, book, etc.).

Syntax:

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

Example:

```
#include <stdio.h>
struct Student {
    int id;
    char name[20];
    float marks;
};
int main() {
    struct Student s1 = {101, "Anu", 89.5};
    printf("Student Details:\n");
    printf("ID: %d\nName: %s\nMarks: %.2f", s1.id, s1.name, s1.marks);
    return 0;
}
```

Output:

```
Student Details:
ID: 101
Name: Anu
Marks: 89.50
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Declaring Structure Variables

- Declaring a structure variable means creating an instance of a structure that allocates memory for all its members.
- Once a structure is defined, we can declare variables of that structure type to store data for each member.

Syntax:

```
struct structure_name variable_name;
```

Types of Structure Variable Declarations

There are three common methods of declaring structure variables in C :

1. Separate Declaration

In this method, the structure is defined first, and then variables are declared separately.

Syntax:

```
struct structure_name {
   data_type member1;
   data_type member2;
};
struct structure_name var1, var2;
```

2. Declaration with Definition

Here, structure definition and variable declaration occur together in a single step.

Syntax:

```
struct structure_name {
   data_type member1;
   data_type member2;
} var1, var2;
```

3. Declaration using typedef

The typedef keyword gives a new name (alias) to a structure, allowing variables to be declared without using the struct keyword.

Syntax:

```
typedef struct {
   data_type member1;
   data_type member2;
} alias_name;
alias_name var1, var2;
```

Dr.Kavitha N
Associate Professor
Dept. of IT

Example:

Program:

```
#include <stdio.h>
// Separate Declaration
struct Student1 {
    int id;
    char name[20];
};
struct Student1 s1;

// Declaration with Definition
struct Student2 {
    int roll;
    float marks;
} s2;

// Using typedef
typedef struct {
    int age;
    char grade;
} Student3;

int main() {
    // Assigning values
    s1.id = 101;
    s2.roll = 202;
    s2.marks = 89.5;
     Student3 s3;   // using typedef type
    s3.age = 18;
    s3.grade = 'A';
     // Displaying results
    printf("Using Separate Declaration:\n");
    printf("ID: %d\n\n", s1.id);

    printf("Using Declaration with Definition:\n");
    printf("Roll: %d\nMarks: %.2f\n\n", s2.roll, s2.marks);

    printf("Using typedef Declaration:\n");
    printf("Age: %d\nGrade: %c\n", s3.age, s3.grade);

    return 0;
}
```

Output:

```
Using Separate Declaration:
ID: 101

Using Declaration with Definition:
Roll: 202
Marks: 89.50

Using typedef Declaration:
Age: 18
Grade: A
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Accessing Structure Members

- Accessing structure members means retrieving or modifying the values stored in each member of a structure variable.
- In C, this is done using the dot (.) operator for normal structure variables and the arrow (->) operator for structure pointers.

Syntax:

Using Dot Operator (.):

structure_variable.member_name;

- Used when we access members of a normal (non-pointer) structure variable.

Using Arrow Operator (->):

structure_pointer->member_name;

- Used when we access members through a pointer to structure.

Output:

Using Dot Operator:
ID: 101, Marks: 89.50

Using Arrow Operator:
ID: 101, Marks: 89.50

Example Program:

```
#include <stdio.h>
struct Student {
    int id;
    float marks;
};
int main() {
    struct Student s1 = {101, 89.5};   // normal structure variable
    struct Student *ptr = &s1;         // pointer to structure
    printf("Using Dot Operator:\n");
    printf("ID: %d, Marks: %.2f\n\n", s1.id, s1.marks);
    printf("Using Arrow Operator:\n");
    printf("ID: %d, Marks: %.2f\n", ptr->id, ptr->marks);
    return 0;
}
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Array of Structures

An Array of Structures is a collection of structure variables of the same type, stored in contiguous memory locations. It allows storing and managing data for multiple entities (like several students, employees, or books) efficiently using a single structure definition.

Example:

```
#include <stdio.h>
struct Student {
    int id;
    float marks;
};
int main() {
    struct Student s[3] = {
        {101, 85.5},
        {102, 90.0},
        {103, 78.5}
    };
    for(int i = 0; i < 3; i++) {
        printf("Student %d -> ID: %d, Marks: %.2f\n", i + 1, s[i].id, s[i].marks);
    }
    return 0;
}
```

Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
};
struct structure_name variable_name[array_size];
```

Output:

```
Student 1 -> ID: 101, Marks: 85.50
Student 2 -> ID: 102, Marks: 90.00
Student 3 -> ID: 103, Marks: 78.50
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Nested Structure in C (Structure within Structure)

A Nested Structure means defining one structure inside another.
It helps represent complex data (like an address inside a student record) in an organized way.

Example:

```
#include <stdio.h>
struct Address {
    char city[20];
    int pincode;
};
struct Student {
    int id;
    struct Address addr;   // nested structure
};
int main() {
    struct Student s1 = {101, {"Hyderabad", 500001}};
    printf("ID: %d\nCity: %s\nPincode: %d\n", s1.id, s1.addr.city, s1.addr.pincode);
    return 0;
}
```

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;
};
```

Output:

```
ID: 101
City: Hyderabad
Pincode: 500001
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Structure as Function Parameter

A structure can be passed as a parameter to a function either by value (copy of structure) or by reference (using pointer). This allows entire sets of related data to be sent to functions conveniently.

Passing Structure by Value

- A copy of the entire structure is passed to the function.
- Any changes made inside the function do not affect the original structure.
- Safer but slightly slower due to copying.

Syntax:
void function_name(struct structure_name variable);

Passing Structure by Reference

- The address of the structure is passed (using pointer).
- Function can directly modify the original structure data.
- Faster, since no copying occurs.

Syntax:
void function_name(struct structure_name *variable);

Dr.Kavitha N
Associate Professor
Dept. of IT

Structure Returning from Function

In C, a function can return a structure as its result. This allows returning multiple related values (like ID, name, and marks) together from one function — instead of using several return statements or global variables.

Syntax:

```
struct structure_name function_name(parameters);
```

When defining :

```
struct structure_name function_name(parameters) {
    // create structure variable
    // assign values
    return variable;
}
```

Example:

```
#include <stdio.h>

struct Student {
    int id;
    float marks;
};
struct Student getStudent() {        // function returning structure
    struct Student s1 = {101, 92.5};
    return s1;                       // returning structure variable
}
int main() {
    struct Student s = getStudent(); // store returned structure
    printf("ID: %d\nMarks: %.2f\n", s.id, s.marks);
    return 0;
}
```

Output:

ID: 101
Marks: 92.50

Dr.Kavitha N
Associate Professor
Dept. of IT

typedef Keyword

The typedef keyword in C is used to create an alias (alternative name) for an existing data type — either a primitive, user-defined, or complex type like a structure.
It improves readability, clarity, and ease of use in large programs.

Syntax:

```
typedef existing_type new_name;
```

for structures:

```
typedef struct {
    data_type member1;
    data_type member2;
} alias_name;
```

Example:

```
#include <stdio.h>
typedef struct {
    int id;
    float marks;
} Student;    // alias name for structure
int main() {
    Student s1 = {101, 88.5};    // no need to write 'struct'
    printf("ID: %d, Marks: %.2f\n", s1.id, s1.marks);
    return 0;
}
```

Output:

```
ID: 101, Marks: 88.50
```

Dr.Kavitha N
Associate Professor
Dept. of IT

UNION

A union in C is a user-defined data type similar to a structure,
but all members share the same memory location.
This means only one member can store a value at a time — saving memory when you don't need all data simultaneously.

Syntax:

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

Advantages of Union :

- Efficient Memory Usage — shares same memory space.
- Useful in Embedded Systems where resources are limited.
- Can store different types at different times.

Disadvantages of Union :

- Only one value can be stored at a time.
- Misuse can cause data corruption.
- Difficult to debug if used incorrectly.

Dr.Kavitha N
Associate Professor
Dept. of IT

Declaring and Initializing Union Variables

Declaration:

```
union Data d1, d2;
```

Initialization (Single Member):

```
union Data d1 = {10};
```

With typedef:

```
typedef union {
    int id;
    char grade;
} Info;

Info student1 = {101};
```

Example:

```
#include <stdio.h>
union Info {
    int id;
    char grade;
};
int main() {
    union Info s1;
    s1.id = 101;
    printf("ID: %d\n", s1.id);

    s1.grade = 'A';
    printf("Grade: %c\n", s1.grade);  // Overwrites id
    return 0;
}
```

Output:

```
ID: 101
Grade: A
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Accessing Union Members

Accessing union members in C is done using the dot (.) operator — similar to structures.
If you use a pointer to a union, then the arrow (→) operator is used.
Since all members share the same memory, only the most recently assigned member holds a valid value.

Syntax:
For normal variable:
union union_name variable_name;
variable_name.member_name;

For pointer variable:
union union_name *ptr;
ptr->member_name;

Example:

```
#include <stdio.h>

union Employee {
    int id;
    float salary;
};

int main() {
    union Employee e1;      // normal union variable
    union Employee *ptr;    // pointer to union
    ptr = &e1;

    // Accessing using dot operator
    e1.id = 101;
    printf("Using dot operator -> ID: %d\n", e1.id);

    // Accessing using arrow operator
    ptr->salary = 55000.75;
    printf("Using arrow operator -> Salary: %.2f\n", ptr->salary);

    return 0;
}
```

Output:

```
Using dot operator -> ID: 101
Using arrow operator -> Salary: 55000.75
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Nested Union

A Nested Union means a union declared inside another union.

It allows grouping related data types that share the same memory within different logical levels — useful when one of several groups of data can be stored at a time.

Syntax:

```
union Outer {
 int id;
 union Inner {
 float marks;
 char grade;
 } inner;
};
```

Example:

```
#include <stdio.h>
union Data {
    int id;
    union Info {
        float marks;
        char grade;
    } info;
};
int main() {
    union Data d;
    d.id = 101;
    d.info.grade = 'A';
    printf("ID: %d\nGrade: %c\n", d.id, d.info.grade);
    return 0;
}
```

Output:

```
ID: 101
Grade: A
```


Dr.Kavitha N
Associate Professor
Dept. of IT

Structure within Union

A structure inside a union means one of the union's members is itself a structure.
It allows the union to hold either a group of structured data or some other type, all sharing the same memory.

Union within Structure

A union inside a structure means one member of the structure is a union.
This allows combining common data (always used) with optional data (used one at a time).

Output:
ID: 101
Score: 95
Grade: A

Example:

```
#include <stdio.h>
struct Marks {
    int score;
};
union Data {
    struct Marks m;   // structure within union
    char grade;
};
struct Student {
    int id;
    union Data info;  // union within structure
};
int main() {
    struct Student s1 = {101};
    s1.info.m.score = 95;
    printf("ID: %d\nScore: %d\n", s1.id, s1.info.m.score);

    s1.info.grade = 'A';
    printf("Grade: %c\n", s1.info.grade);

    return 0;
}
```

Dr.Kavitha N
Associate Professor
Dept. of IT

Dynamic Memory Allocation (DMA)

Dynamic Memory Allocation (DMA) allows memory to be allocated or freed at runtime (while the program is running), instead of at compile-time.
It provides flexibility and efficient use of memory.

malloc() — Memory Allocation

- malloc() stands for Memory Allocation.
- It allocates a single continuous block of memory in the heap at runtime.
- The memory contains garbage (uninitialized) values.

Syntax:

```
ptr = (type*) malloc(size_in_bytes);
```

Example:

```
int *p = (int*) malloc(3 * sizeof(int));
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(3 * sizeof(int));
    for(int i=0;i<3;i++) p[i]=i+1;
    for(int i=0;i<3;i++) printf("%d ", p[i]);
    free(p);
}
```

Output:

1 2 3

Dr.Kavitha N
Associate Professor
Dept. of IT

calloc() — Contiguous Allocation

calloc() stands for Contiguous Allocation.
It allocates multiple blocks of memory and initializes all bytes to zero (0).

Syntax:

```
ptr = (type*) calloc(num_elements, size_of_each);
```

Example:

```
int *p = (int*) calloc(3, sizeof(int));
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) calloc(3, sizeof(int));
    for(int i=0;i<3;i++) printf("%d ", p[i]);
    free(p);
}
```

Output:

```
0 0 0
```


Dr.Kavitha N
Associate Professor
Dept. of IT

realloc() — Reallocation

realloc() stands for Reallocation of Memory. It is used to resize previously allocated memory (by malloc() or calloc()) without losing existing data.

Syntax:

ptr = (type*) realloc(ptr, new_size_in_bytes);

Example:

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

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(3 * sizeof(int));
    for(int i=0;i<3;i++) p[i]=i+1;
    p = (int*) realloc(p, 5 * sizeof(int));
    for(int i=3;i<5;i++) p[i]=i+1;
    for(int i=0;i<5;i++) printf("%d ", p[i]);
    free(p);
}
```

Output:

1 2 3 4 5

Dr.Kavitha N
Associate Professor
Dept. of IT

free() — Memory Deallocation

free() function is used to release previously allocated memory back to the system. Prevents memory leaks and ensures efficient memory use.

Syntax:

```
free(ptr);
```

Example:

```
int *p = (int*) malloc(5 * sizeof(int));
free(p);
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int *p = (int*) malloc(3 * sizeof(int));
    free(p);
    p = NULL; // good practice
}
```


Dr.Kavitha N
Associate Professor
Dept. of IT