Unit 3

Structures, Unions, Pointers and Memory Management

Learning Objectives

- To understand and use structures and unions for grouped data storage
- To declare, manipulate, and apply pointers for efficient memory access
- To work with arrays, structures, and functions using pointers
- To implement dynamic memory allocation using standard library functions
- To identify and apply appropriate storage classes and avoid pointer-related errors

Structure

3.1 Structures in C

In C, a **structure** (struct) is a user-defined data type that allows grouping of variables of **different types** under a single name. It is used to model real-world entities like a Student, Employee, or Book.

3.1.1 Definition and Syntax of Structures

Definition:

A structure is defined using the struct keyword, followed by a name and a block containing member variables.

Syntax:

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

Example:

```
struct Student {
    int roll_no;
    char name[30];
    float marks;
};
```

This defines a structure named Student with three members: roll_no, name, and marks.

3.1.2 Declaring and Accessing Members

Once the structure is defined, you can declare variables of that structure type.

Declaration:

```
struct Student s1;
```

Accessing Members:

Use the **dot operator (.)** to access members:

```
s1.roll_no = 101;
s1.marks = 85.5;
strcpy(s1.name, "Amit");
```

Printing Values:

```
printf("Roll No: %d\n", s1.roll_no);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);
```

3.1.3 Initializing Structures

You can initialize structure variables at the time of declaration.

Example:

```
struct Student s2 = {102, "Reena", 92.0};
```

You can also assign member-wise values:

```
s2.roll_no = 103;
strcpy(s2.name, "Karan");
s2.marks = 78.5;
```

From C99 onwards, you can use **designated initializers**:

```
struct Student s3 = {.name = "Sneha", .roll_no = 104, .marks = 88.2};
```

3.1.4 Array of Structures

To store data for multiple entities (e.g., many students), use an **array of structures**.

Example:

```
struct Student students[3];
```

You can then use a loop to input/output data:

```
for (int i = 0; i < 3; i++) {
    scanf("%d", &students[i].roll_no);
    scanf("%s", students[i].name);
    scanf("%f", &students[i].marks);
}
```

Loop to display:

```
for (int i = 0; i < 3; i++) {
    printf("%d %s %.2f\n", students[i].roll_no, students[i].name, students[i].marks);
}
```

3.1.5 Nested Structures and Complex Structures

Structures can contain other structures as members. This is useful to represent hierarchical data.

Example:

```
struct Date {
    int day, month, year;
};

struct Student {
    int roll_no;
    char name[30];
    struct Date dob;    // Nested structure
};
```

Access nested members using dot (.) multiple times:

```
struct Student s1;
s1.dob.day = 15;
s1.dob.month = 8;
s1.dob.year = 2003;
```

Complex Structures Example:

```
struct Address {
    char city[20];
    int pin;
};

struct Employee {
    int emp_id;
    char name[30];
    struct Address addr;
};
```

This allows more detailed and structured representation of real-world data.

3.2 Unions and Memory Sharing

A **union** in C is a special data type similar to a structure, but with one major difference: **all members of a union share the same memory location**. This makes unions memory-efficient when you only need to store **one value at a time** out of several possibilities.

3.2.1 Definition and Syntax of Unions

Definition:

A union groups different variables (members) under one name, but **only one member can store a value at a time** because they share the same memory location.

Syntax:

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

Example:

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

You can declare a union variable:

```
union Data d1;
```

3.2.2 Comparison with Structures

Feature	Structure (struct)	Union (union)
Memory	Allocates memory for **all members**	Allocates memory for **largest member only**
Member Access	All members can hold values simultaneously	Only **one member** holds a value at a time
Use Case	Store related data with different types	Store **one of many types**, save memory
Syntax	Similar	Similar

Example:

```
struct S {
    int x;
    float y;
};

union U {
    int x;
    float y;
};
```

Memory:

- struct S → uses memory for int + float
- union U → uses memory for **whichever is larger** (int or float)

3.2.3 Memory Allocation in Unions

In a union, **only one memory block is allocated** that is **large enough to hold the biggest member**.

Example:

```
union Test {
    int i;          // 4 bytes
    float f;        // 4 bytes
    char str[20]; // 20 bytes
};
```

- Total memory used: **20 bytes**
- All members share the **same 20-byte space**
- Changing one member **overwrites** the others

Accessing:

```
union Test t;
t.i = 10;
printf("%d\n", t.i);          // Valid

t.f = 3.14;
printf("%f\n", t.f);          // Overwrites i

strcpy(t.str, "Hello");
printf("%s\n", t.str);       // Overwrites f
```

3.2.4 Use Cases for Unions

Unions are used when:

1. **Memory Efficiency** is important (e.g., embedded systems)
2. You need to store **only one value** out of several options at a time
3. You're dealing with **variant data types**

Examples:

- **Embedded Systems:** Limited RAM, need compact data representation
- **Protocol Parsing:** Reading a data packet that could be an int, char, or float
- **Type Conversion:** Reinterpreting data (e.g., accessing float as bytes)

Example:

```
union Packet {
```

```
    int int_data;
    float float_data;
    char byte[4];
};
```

3.2.5 Unions within Structures

A union can be a **member of a structure**, often used to store variant data types **with a type indicator**.

Example:

```
struct Value {
    int type;    // 1 = int, 2 = float
    union {
        int i;
        float f;
    } data;
};
```

Usage:

```
struct Value v;
v.type = 1;
v.data.i = 42;

if (v.type == 1)
    printf("Integer: %d\n", v.data.i);
```

This pattern is useful in **interpreters**, **message systems**, and **data parsers**.

3.3 Arrays of Structures

When you need to store and manage **multiple records** of the same structured data type (like students, employees, books), you use an **array of structures**. This helps in logically organizing and processing multiple data entries of the same kind.

3.3.1 Declaring Structure Arrays

You first define the structure, and then declare an array of that structure type.

Example:

```
struct Student {
    int roll_no;
    char name[30];
    float marks;
};

struct Student students[5];    // Array of 5 Student records
```

Each element students[0], students[1], ... is a separate Student variable.

3.3.2 Accessing Elements using Indexing

Access individual structure elements using array indexing and the dot (.) operator.

Example:

```
students[0].roll_no = 101;
```

```
strcpy(students[0].name, "Rahul");
students[0].marks = 85.5;
```
This sets the values for the **first student** in the array.

3.3.3 Looping through Structure Arrays

To handle multiple records, use a loop.

Input Example:

```
for (int i = 0; i < 5; i++) {
    printf("Enter roll no, name, and marks for student %d:\n", i + 1);
    scanf("%d", &students[i].roll_no);
    scanf("%s", students[i].name);
    scanf("%f", &students[i].marks);
}
```

Output Example:

```
for (int i = 0; i < 5; i++) {
    printf("%d %s %.2f\n", students[i].roll_no, students[i].name, students[i].marks);
}
```

3.3.4 Use in Sorting and Searching Records

Sorting Example (By Marks - Bubble Sort):

```
for (int i = 0; i < 5 - 1; i++) {
    for (int j = 0; j < 5 - i - 1; j++) {
        if (students[j].marks > students[j + 1].marks) {
            struct Student temp = students[j];
            students[j] = students[j + 1];
            students[j + 1] = temp;
        }
    }
}
```

Searching Example (By Roll Number):

```
int target = 103;
for (int i = 0; i < 5; i++) {
    if (students[i].roll_no == target) {
        printf("Found: %s with marks %.2f\n", students[i].name,
students[i].marks);
        break;
    }
}
```

These operations make structure arrays useful in **student record systems**, **inventory management**, and **employee databases**.

3.3.5 Passing Structure Arrays to Functions

Structure arrays can be passed to functions using array notation.

Function Declaration:

```
void display(struct Student s[], int n);
```

Function Definition:

```
void display(struct Student s[], int n) {
```

```
    for (int i = 0; i < n; i++) {
        printf("%d %s %.2f\n", s[i].roll_no, s[i].name, s[i].marks);
    }
}
```

Calling the Function:

```
display(students, 5);
```

This approach improves code modularity and reusability when working with multiple records.

3.4 Introduction to Pointers

A **pointer** is a special variable in C that stores the **memory address** of another variable. Pointers are essential for efficient memory manipulation, dynamic memory allocation, and working with arrays, functions, and structures.

3.4.1 Pointer Declaration and Initialization

Syntax:

```
data_type *pointer_name;
```

The * indicates that the variable is a **pointer** to a certain data type.

Example:

```
int a = 10;
int *p;          // Declaring pointer to int
p = &a;          // Storing address of a in p
```

You can also initialize during declaration:

```
int *p = &a;
```

3.4.2 Dereferencing and Address-of Operator

Address-of Operator (&):

- Used to get the **memory address** of a variable.

```
int x = 5;
printf("%p", &x);    // Outputs address of x
```

Dereference Operator (*):

- Used to **access the value** stored at the address the pointer is pointing to.

```
int x = 5;
int *p = &x;
printf("%d", *p);    // Outputs 5 (value at address stored in p)
```

3.4.3 Pointer to Variables

Pointers can be used to modify the actual variable by accessing its memory address.

Example:

```
void update(int *num) {
    *num = *num + 10;
}

int main() {
    int a = 5;
```

```
    update(&a);
    printf("%d", a);    // Output: 15
}
```
This is called **pass-by-reference**.

3.4.4 Pointer Expressions and Precedence

You can perform arithmetic operations on pointers and use them in expressions.

Valid Pointer Operations:

- Increment/decrement: p++, p--
- Addition/subtraction: p + n, p - n
- Comparison: p1 == p2, p1 != p2

Example:

```
int arr[] = {10, 20, 30};
int *p = arr;

printf("%d", *(p + 1));    // Output: 20
```

Precedence:

The * (dereference) and & (address-of) have **lower precedence** than some operators, so parentheses may be needed.

```
*ptr + 1       // adds 1 to the value pointed by ptr
*(ptr + 1)     // accesses next element in memory
```

3.4.5 Null and Wild Pointers

Null Pointer:

A pointer that **points to nothing** (i.e., address 0).

```
int *ptr = NULL;
```

- Use to indicate that a pointer is **not yet assigned**
- Helps prevent **dangling pointer errors**

Wild Pointer:

A pointer that has **not been initialized** and points to an unknown memory location.

```
int *ptr;          // Wild pointer (dangerous)
*ptr = 10;         // May crash the program
```

Best Practice: Always initialize pointers to NULL if not in use.

3.5 Pointer Arithmetic

Pointers in C not only store addresses but can also participate in **arithmetic operations**. These operations are useful for navigating arrays, data buffers, and structures in memory.

3.5.1 Arithmetic Operations on Pointers

You can perform certain arithmetic operations on pointers:

- **Addition (+)**

- **Subtraction (-)**
- **Increment (++)**
- **Decrement (--)**
- **Pointer Difference (ptr2 - ptr1)**

These operations depend on the **data type** the pointer points to.
Example:

```
int arr[] = {10, 20, 30, 40};
int *p = arr;

printf("%d\n", *(p + 2));    // Outputs: 30
```

- p + 2 means move 2 integers ahead (not 2 bytes, but 2 * sizeof(int) bytes)

3.5.2 Pointer Increment and Decrement

Incrementing a pointer moves it to the **next memory location of its type**.
Example:

```
int arr[] = {5, 10, 15};
int *p = arr;

printf("%d\n", *p);      // Output: 5
p++;                            // Move to next element
printf("%d\n", *p);      // Output: 10
```

Decrement Example:

```
p--;    // Move back to previous element
```

You should **not increment/decrement** a pointer **beyond its array bounds** (undefined behavior).

3.5.3 Pointer Comparisons

Pointers can be compared using:

- ==, != (equality)
- <, >, <=, >= (relative position)

Example:

```
int arr[] = {2, 4, 6};
int *p1 = &arr[0];
int *p2 = &arr[2];

if (p1 < p2) {
      printf("p1 is before p2 in memory\n");
}
```

Pointer comparisons are mainly valid when both pointers point to elements **within the same array**.

3.5.4 Pointer and Array Equivalence

Arrays and pointers are **closely related** in C. An array name acts like a pointer to its first element.

Example:

```
int arr[] = {10, 20, 30};
int *p = arr;

printf("%d\n", *arr);       // 10
printf("%d\n", *(arr + 1)); // 20
printf("%d\n", p[2]);       // 30
```

- arr[i] is equivalent to *(arr + i)
- p[i] is equivalent to *(p + i)

Note: Arrays are **not pointers**, but they can decay into pointers when passed to functions.

3.5.5 Type Casting with Pointers

Type casting allows you to convert one pointer type to another. It is useful when working with generic memory (like in void *) or handling raw byte data.

Syntax:

```
int x = 100;
void *vp = &x;
printf("%d\n", *( (int *)vp ));
```

Use Case:

Casting void * in dynamic memory:

```
void *ptr = malloc(10 * sizeof(int));
int *iptr = (int *)ptr;
```

Be cautious with type casting:

- Always ensure proper alignment and type safety.
- Incorrect casting can lead to **undefined behavior** or **crashes**.

3.6 Arrays with Pointers

Pointers and arrays are tightly related in C. Pointers can be used to access and manipulate arrays efficiently, and understanding their relationship enables advanced memory operations and flexible code.

3.6.1 Accessing Array Elements Using Pointers

In C, the name of an array (e.g., arr) can be used as a pointer to its first element.

Example:

```
int arr[5] = {10, 20, 30, 40, 50};
int *ptr = arr;    // Equivalent to int *ptr = &arr[0];

printf("%d\n", *(ptr + 2));    // Outputs: 30
```

- arr[i] is the same as *(arr + i)

- ptr[i] is the same as *(ptr + i)

This is helpful when iterating through an array using pointer arithmetic.

3.6.2 Pointer to an Array

A pointer to an entire array (not just the first element) is different from a pointer to the first element.

Syntax:

```
int arr[3] = {1, 2, 3};
int (*p)[3] = &arr;
```

- p is a pointer to an array of 3 integers
- Accessing:

```
printf("%d\n", (*p)[1]);    // Outputs: 2
```

This is used in functions that need to refer to the whole array as a single unit.

3.6.3 Array of Pointers

You can create an array where **each element is a pointer**. This is especially useful when:

- Storing multiple strings
- Pointing to dynamically allocated memory blocks

Example:

```
int a = 5, b = 10, c = 15;
int *arr[3];

arr[0] = &a;
arr[1] = &b;
arr[2] = &c;

printf("%d\n", *arr[1]);    // Outputs: 10
```

Each arr[i] is a pointer to an integer.

3.6.4 String Arrays using Pointers

Instead of using 2D arrays for strings, you can use an **array of character pointers** for better flexibility.

Example:

```
char *names[] = {"Alice", "Bob", "Charlie"};

printf("%s\n", names[0]);    // Outputs: Alice
```

- Each element is a pointer to the first character of a string.
- This saves space compared to char names[3][20] and allows different-length strings.

3.6.5 Multidimensional Arrays and Pointers

Pointers can also be used with 2D arrays. Understanding memory layout is key.

Example:

```
int arr[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
};

int *p = &arr[0][0];

for (int i = 0; i < 6; i++) {
        printf("%d ", *(p + i));    // Outputs: 1 2 3 4 5 6
}
```

Pointer to 2D Array:

```
int (*p2D)[3] = arr;
printf("%d\n", p2D[1][2]);    // Outputs: 6
```

- p2D is a pointer to an array of 3 integers (a row in the 2D array)
- Helps pass 2D arrays to functions accurately

3.7 Pointers and Structures

Pointers are commonly used with structures in C to create flexible and memory-efficient programs. Using pointers with structures enables dynamic memory allocation, efficient function calls, and the creation of complex data types like linked lists.

3.7.1 Pointer to Structure

You can declare a pointer that points to a structure, just like you do for basic data types.

Syntax:

```
struct Student {
        int roll_no;
        float marks;
};

struct Student s1;
struct Student *ptr = &s1;
```

Here, ptr is a pointer to a Student structure.

3.7.2 Accessing Members with Arrow (->) Operator

When you have a **pointer to a structure**, you use the -> operator to access its members (instead of the dot . operator used with normal structure variables).

Example:

```
ptr->roll_no = 101;
ptr->marks = 88.5;

printf("%d %.2f\n", ptr->roll_no, ptr->marks);
```

- ptr->roll_no is equivalent to (*ptr).roll_no

This is more convenient and readable when using structure pointers.

3.7.3 Passing Structures by Pointer to Functions

Passing the **address of a structure** to a function is more memory-efficient than passing the whole structure.

Example:

```
void display(struct Student *s) {
    printf("Roll: %d, Marks: %.2f\n", s->roll_no, s->marks);
}

int main() {
    struct Student s1 = {102, 91.0};
    display(&s1);    // Pass by pointer
    return 0;
}
```

This avoids copying the entire structure and allows the function to **modify** the original data.

3.7.4 Dynamic Structures using Pointers

You can dynamically allocate memory for structures using malloc() or calloc().

Example:

```
struct Student *s;
s = (struct Student *)malloc(sizeof(struct Student));

s->roll_no = 105;
s->marks = 87.6;
```

This allows creating structure instances **at runtime**, which is useful when the number of records is not known in advance.

Remember to free the memory:

```
free(s);
```

3.7.5 Structures within Pointers (Linked Structure Concept)

You can define a structure that **contains a pointer to another structure of the same type**. This concept is used to build **linked data structures** like **linked lists**, **trees**, and **graphs**.

Example:

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

Here, each Node contains an integer and a pointer to the **next node** in the list.

Creating Nodes:

```
struct Node *head = (struct Node *)malloc(sizeof(struct Node));
head->data = 10;
head->next = NULL;
```

This allows you to **dynamically link** multiple structures together to form chains.

3.8 Dynamic Memory Allocation

In C, **dynamic memory allocation** allows you to allocate memory **at runtime** instead of at compile-time. This is essential when the size of data (like arrays or structures) is not known in advance or needs to change during program execution.
C provides a set of standard library functions (malloc(), calloc(), realloc(), free()) from the <stdlib.h> header for dynamic memory management.

3.8.1 Introduction to Dynamic Memory

In **static memory allocation**, memory is fixed and allocated during compile time.
In **dynamic memory allocation**, memory is requested from the **heap** during runtime using pointers. This gives more flexibility and better memory efficiency.
Key functions:

- malloc() — allocate a single block
- calloc() — allocate multiple blocks and initialize them to zero
- realloc() — resize memory block
- free() — deallocate memory

3.8.2 malloc() and Memory Allocation

malloc() stands for **memory allocation**. It allocates a block of memory of a specified size (in bytes) and returns a pointer to the **beginning** of the block.
Syntax:

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

Example:

```
int *arr;
arr = (int *)malloc(5 * sizeof(int));

if (arr == NULL) {
      printf("Memory allocation failed");
}
```

- The allocated memory is **not initialized** (contains garbage values).
- Always check if malloc() returned NULL.

3.8.3 calloc() for Multiple Block Allocation

calloc() stands for **contiguous allocation**. It allocates multiple blocks of memory and initializes all bytes to zero.
Syntax:

```
ptr = (cast_type *) calloc(num_elements, size_of_each_element);
```

Example:

```
int *arr;
arr = (int *)calloc(5, sizeof(int));    // allocates and initializes to 0
```

Compared to malloc(), calloc() initializes memory, which is useful when you need zeroed arrays or structures.

3.8.4 realloc() for Resizing Memory

realloc() is used to **resize** a previously allocated memory block. It keeps the existing data (as much as fits) and may move the block to a new location if needed.

Syntax:

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

Example:

arr = (int *)realloc(arr, 10 * sizeof(int));

- If new size is larger, the added memory is uninitialized.
- If reallocation fails, it returns NULL and the original block remains unchanged.

3.8.5 free() and Deallocating Memory

free() is used to **release** the dynamically allocated memory and return it to the system.

Syntax:

free(ptr);

Example:

free(arr);

- After freeing, set the pointer to NULL to avoid **dangling pointer** issues.
- Never free() memory that was not allocated dynamically.

Common Tips:

- Always check if the pointer returned by malloc() or calloc() is not NULL
- Do not use memory after it has been freed
- Use sizeof() to calculate memory size instead of hardcoding values
- Memory leaks happen when dynamically allocated memory is not freed

3.9 Storage Classes in C

A **storage class** in C defines **four things** about a variable:

1. **Scope** – where the variable is accessible
2. **Lifetime** – how long the variable exists in memory
3. **Default Initial Value** – if uninitialized, what value it starts with
4. **Location** – where it's stored (RAM, CPU register, etc.)

There are **four storage classes** in C:

- auto

- static
- register
- extern

3.9.1 Overview of Storage Classes

Storage Class	Scope	Lifetime	Default Value	Location
auto	Local	Block (function)	Garbage value	Memory (stack)
static	Local/Global	Entire program	Zero (if not initialized)	Memory (data segment)
register	Local	Block	Garbage value	CPU register (if available)
extern	Global	Entire program	Value from linked file	Memory

3.9.2 auto and Local Variables

- auto is the **default** storage class for local variables.
- You rarely need to explicitly write auto.

Example:

```
void func() {
    auto int a = 10;    // same as: int a = 10;
    printf("%d", a);
}
```

- Scope: Within the function/block
- Lifetime: Exists until function exits

3.9.3 static and Persistent Variables

- A static variable **retains its value** between multiple function calls.
- For global variables, static **restricts access** to the same file (internal linkage).

Function-Level Example:

```
void countCalls() {
    static int count = 0;
    count++;
    printf("Called %d times\n", count);
}
```

- Output increases every time function is called.
- Lifetime: Entire program

- Scope: Local to function

File-Level Example:

```
static int fileVar = 5;    // cannot be accessed from other files
```

3.9.4 register for Faster Access

- register tells the compiler to store the variable in a **CPU register** instead of RAM.
- Used for frequently accessed variables (e.g., loop counters).

Example:

```
void loop() {
    register int i;
    for (i = 0; i < 1000; i++) {
        // faster access
    }
}
```

Limitations:

- You **cannot get the address** of a register variable (no & operator).
- Compiler **may ignore** the request if registers are full.

3.9.5 extern and Global Variable Access

- extern is used to declare a global variable that is **defined in another file** or scope.
- It tells the compiler: “the variable exists somewhere else.”

File 1 (main.c):

```
int num = 10;    // definition
```

File 2 (utils.c):

```
extern int num;    // declaration
printf("%d", num);
```

- Used for **sharing global variables** across multiple files.
- Cannot initialize extern variables during declaration.

3.10 Pointer Pitfalls and Best Practices

Pointers offer powerful control over memory, but they also introduce risks. Misusing pointers can lead to **crashes**, **undefined behavior**, or **security vulnerabilities**. This section highlights common pointer problems and how to avoid them.

3.10.1 Dangling Pointers

A **dangling pointer** refers to a pointer that continues to point to a memory location **after the memory has been freed** or the variable has gone out of scope.

Example:

```
int *ptr = (int *)malloc(sizeof(int));
*ptr = 100;
free(ptr);          // Memory is freed
*ptr = 200;         // Undefined behavior – ptr is dangling
```

Also happens when returning the address of a **local variable**:

```
int* getPointer() {
    int x = 10;
    return &x;    // x goes out of scope after function ends
}
```

How to Avoid:

- Set pointer to NULL after free()

```
free(ptr);
ptr = NULL;
```

3.10.2 Memory Leaks and Detection

A **memory leak** occurs when memory is allocated using malloc() or calloc() but **not freed**. Over time, this reduces available memory and degrades performance.

Example:

```
int *ptr = (int *)malloc(100 * sizeof(int));
// forgot to call free(ptr)
```

Detection Tools:

- Use **Valgrind** (Linux)
- Use memory analysis tools in IDEs
- Manually track allocations and deallocations

How to Avoid:

- Always call free() when memory is no longer needed
- Avoid losing references to dynamically allocated memory

3.10.3 Uninitialized Pointers

An uninitialized pointer contains a **garbage address** and accessing it causes undefined behavior or program crashes.

Example:

```
int *ptr;           // uninitialized
*ptr = 10;          // dangerous
```

How to Avoid:

- Initialize all pointers to NULL

```
int *ptr = NULL;
```

- Only use pointer after assigning a valid memory address

3.10.4 Pointer Type Mismatch

Pointer type mismatch happens when you assign a pointer of one type to another without a proper cast, which can lead to **wrong memory interpretation**.

Example:

```
float f = 3.14;
int *iptr = (int *)&f;    // forced cast (unsafe)
printf("%d", *iptr);       // accessing float as int – undefined
```

Avoid This By:

- Using proper pointer types
- Avoiding unsafe casts unless absolutely necessary
- Ensuring pointer arithmetic is done with correct data types

3.10.5 Best Practices in Pointer Usage

To use pointers safely and effectively:

- **Initialize pointers** to NULL
- Always **check memory allocation**:

```
ptr = malloc(...);
if (ptr == NULL) {
      // handle allocation failure
}
```

- **Use const** with pointer parameters if the data should not be modified

```
void printData(const int *data) {
      printf("%d", *data);
}
```

- Avoid pointer arithmetic unless necessary
- Prefer **array notation** for clarity when working with arrays
- **Document pointer usage** clearly in code comments
- **Use memory management tools** to detect leaks and misuse

3.11 Summary

This module introduced advanced concepts in C programming that are critical for writing efficient and modular programs. It started with structures and unions, allowing grouping of different types of data under a single unit. Pointers were extensively covered — from basic declarations to advanced topics like pointer arithmetic, dynamic memory allocation, and working with pointers in structures and arrays. Special focus was given to memory handling, storage classes, and pitfalls to avoid with pointers. By mastering these topics, students can optimize data handling, understand memory-level operations, and write more flexible, reusable, and performance-oriented code.

3.12 Keywords

Keyword	Definition
struct	Used to define a structure — a collection of related variables
union	Similar to structure but with shared memory for all members
pointer	A variable that stores the memory address of another variable
malloc()	Allocates memory dynamically and returns a pointer to the block
calloc()	Allocates and initializes multiple memory blocks
realloc()	Resizes an already allocated memory block
free()	Deallocates dynamically allocated memory
static	Retains value across function calls and limits scope to file/block
extern	Refers to a global variable declared in another file
register	Requests storage of a variable in a CPU register for faster access
NULL	A constant indicating a pointer points to no valid memory
dangling pointer	A pointer that refers to a memory location that has been freed

3.13 Self-Assessment Questions (Subjective & Case-Based)

Subjective Questions:

1. Define a structure in C. How is it different from a union?
2. Explain how to use dynamic memory allocation with structures.
3. Discuss the use of pointer arithmetic with arrays.
4. What are storage classes in C? Explain each with examples.
5. Compare and contrast malloc(), calloc(), and realloc().

Case-Based Questions:

Case Study 1:

You are building a student record management system that stores the following data for each student: Roll Number, Name, Marks in 3 subjects.

- Design the structure.
- Write functions to input and display records.
- Use an array of structures and pointer-based iteration.

Case Study 2:

A developer has written a program that crashes during execution. The investigation shows a pointer was dereferenced after freeing.

- Identify the problem.
- Suggest modifications to avoid such issues.
- Propose best practices for safe pointer usage in large projects.

3.14 Case Study

Title: Memory-Efficient Contact Management System
A small embedded application needs to store and manage contact information (name, mobile number, email). The size of data is unknown at compile time and needs to be allocated dynamically. Each contact is managed using a structure. The system must:

- Dynamically add and remove contacts.
- Traverse the contacts using pointers.
- Maintain separation of logic using functions.

Approach:

- Define a struct Contact.
- Use malloc() to add new contacts.
- Store contacts in an array of pointers to structures.
- Use functions for display, add, delete.
- Handle memory deallocation using free().

Learning Outcome:
This case demonstrates practical use of structures, pointers, dynamic memory, and modular programming in a real-world scenario.

3.15 References

1. E. Balagurusamy, **"Programming in ANSI C"**, McGraw-Hill Education
2. B. Kernighan & D. Ritchie, **"The C Programming Language"**, Prentice Hall
3. Reema Thareja, **"Programming in C"**, Oxford University Press
4. Dennis M. Ritchie and Brian W. Kernighan, **“C Language Reference Manual”**
5. TutorialsPoint. (2023). C Programming Tutorial
6. GeeksforGeeks. (2023). C Programming Language