Structure, Union, and Dynamic Memory Allocation

Mohammad Maksood Akhter, PhD
Assistant Professor

Department of Computer Science and Engineering

October 30, 2025

Overview

This lecture introduces user-defined data structures and dynamic memory management in C.

- Structures – combining multiple data types
- Unions – sharing memory among members
- Dynamic Memory Allocation (DMA) – using `malloc()`, `calloc()`, `realloc()`, and `free()`

Goal

Understand how to organize data efficiently and allocate memory dynamically.

Structure Concept

Structure Definition

A structure groups variables of different data types under one name.

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

All members have their own memory; total size is sum of all member sizes.

Structure Example

```c
#include <stdio.h>

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

int main() {
    struct Student s1 = {101, "Rahul", 88.5};

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

    return 0;
}
```

Key Points about Structures

- Members are accessed using the dot operator: `object.member`
- Memory allocated = sum of members + alignment padding
- Structures can be passed to functions and returned as values
- Useful for grouping related data (e.g., student, employee)

Union Concept

Union Definition

A union stores different data types in the same memory location.

```c
union Data {
    int i;
    float f;
    char ch;
};
```

Only one member can hold a valid value at a time.

Union Example

```c
#include <stdio.h>

union Data {
    int i;
    float f;
    char ch;
};

int main() {
    union Data d;
    d.i = 10;
    printf("i = %d\n", d.i);
    d.f = 3.14;
    printf("f = %.2f\n", d.f);
    d.ch = 'A';
    printf("ch = %c\n", d.ch);
    return 0;
}
```

Structure vs Union

Feature	Structure	Union
Memory	Each member gets its own memory	Shared memory among all members
Size	Sum of all members	Size of largest member
Usage	All members usable simultaneously	Only one valid at a time
Use Case	Complex data grouping	Memory-efficient representation

Dynamic Memory Allocation (DMA)

Concept

DMA allows allocation of memory at runtime.

- `malloc()` – allocates uninitialized memory
- `calloc()` – allocates and initializes to zero
- `realloc()` – resizes allocated memory
- `free()` – releases allocated memory

DMA helps handle variable-sized data efficiently during runtime.

malloc() Example

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

int main() {
    int *ptr, n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) malloc(n * sizeof(int));

    for(int i=0; i<n; i++)
    scanf("%d", &ptr[i]);

    printf("You entered: ");
    for(int i=0; i<n; i++)
    printf("%d ", ptr[i]);

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

realloc() Example

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

int main() {
    int *ptr = malloc(2 * sizeof(int));
    ptr[0] = 10;
    ptr[1] = 20;

    ptr = realloc(ptr, 4 * sizeof(int));
    ptr[2] = 30;
    ptr[3] = 40;

    for(int i=0; i<4; i++)
    printf("%d ", ptr[i]);

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

Common Mistakes in DMA

- Forgetting to call `free()` $\rightarrow$ memory leak
- Using uninitialized or freed pointers $\rightarrow$ segmentation fault
- Not checking if allocation succeeded (`ptr == NULL`)

Always validate memory allocation before use and free it when done.

Linked List using Structure (1)

Linked lists are built using structures where each node stores data and a pointer to the next node.

```c
#include <stdio.h>
#include <stdlib.h>  // gives proper declaration of malloc, free, calloc, realloc
struct Node {
    int data;
    struct Node *next;
};

int main() {
    struct Node *head = NULL,
    *second = NULL,
    *third = NULL;

    // Allocate memory dynamically
    head = (struct Node*) malloc(sizeof(struct Node));
    second = (struct Node*) malloc(sizeof(struct Node));
    third = (struct Node*) malloc(sizeof(struct Node));
```

```c
head->data = 10;
head->next = second;

second->data = 20;
second->next = third;

third->data = 30;
third->next = NULL;

// Traverse list
struct Node *ptr = head;
while(ptr != NULL) {
    printf("%d -> ", ptr->data);
    ptr = ptr->next;
}

// Free memory
free(head);
free(second);
free(third);
return 0;
```

Linked List Breakdown

- Each node is a **structure** containing data and a pointer to the next node.
- Memory is **dynamically allocated** using `malloc()`.
- The last node's `next` pointer is set to `NULL`.
- Linked lists allow **efficient insertion/deletion** but not random access.

Takeaway: Structures + DMA in Action

Linked Lists combine both concepts — **structures** define node layout, and **dynamic memory allocation** allows flexible memory usage at runtime.

File Handling Concept

Definition

File handling in C allows programs to store and retrieve data permanently from secondary storage (like hard disk).

Unlike variables which lose data when a program ends, files preserve information for later use.

- All file operations are performed using a **FILE pointer**.
- Syntax: `FILE *fp;`

File Operations Overview

- `fopen()` – Opens a file in given mode
- `fprintf()` / `fscanf()` – Write/Read formatted data
- `fgetc()` / `fputc()` – Read/Write single characters
- `fgets()` / `fputs()` – Read/Write strings
- `fclose()` – Closes the opened file

Common modes: `"r"` = read, `"w"` = write (overwrite), `"a"` = append, `"r+"` = read-/update, `"w+"` = write/update.

File Write Example

```c
#include <stdio.h>

int main() {
 FILE *fp;
 fp = fopen("data.txt", "w");

 if(fp == NULL) {
    printf("File not created!\n");
    return 1;
    }

 fprintf(fp, "Hello File Handling!\n");
 fprintf(fp, "C Programming Example\n");

 fclose(fp);
 printf("Data written successfully.");
 return 0;
}
```

Explanation:

- Opens (or creates) file `data.txt` in write mode.
- `fprintf()` writes formatted data into file.
- Always check for `NULL` pointer before writing.
- `fclose()` is mandatory to save changes.

File Read Example

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char str[50];

    fp = fopen("data.txt", "r");
    if(fp == NULL) {
        printf("File not found!\n");
        return 1;
    }

    while(fgets(str, 50, fp) != NULL)
    printf("%s", str);

    fclose(fp);
    return 0;
}
```

Explanation:

- Opens file `data.txt` in read mode.
- `fgets()` reads one line at a time.
- Prints the content to standard output.
- Closes the file at end.

```c
#include <stdio.h>
int main() {
    FILE *fp;
    char str[100];
    fp = fopen("data.txt", "w+");   // Open file for both reading and writing
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    printf("Enter some text: ");         // Take input from user
    fgets(str, sizeof(str), stdin);  // Reads a line of input from the user
    fprintf(fp, "%s", str);  // Write input to the file
    rewind(fp);                // Rewind file pointer to the beginning of the file
    // Read the content back from the file and print it
    printf("\nReading from file:\n");
    while (fgets(str, sizeof(str), fp) != NULL) {
        printf("%s", str);
    }
    fclose(fp);
    return 0;
}
```

Key Points on File Handling

- Always verify that `fopen()` succeeded before file operations.
- Use `fclose()` to release file resources.
- `fprintf()` and `fscanf()` work like `printf()` and `scanf()` but on files.
- Avoid overwriting by using append mode (`"a"`).
- Combine with structures to store records like student details.

File handling bridges memory and storage — making programs capable of saving data permanently.

Thank You!

Questions?