Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

STRUCTURES AND UNIONS IN C

CHAPTER 1: INTRODUCTION TO STRUCTURES AND UNIONS

Structures and unions are two important **user-defined data types** in C that allow programmers to store multiple related variables in a **single unit**. These data types enhance program efficiency, improve data organization, and facilitate complex data manipulations.

Why Use Structures and Unions?

1. **Data Organization:** Structures and unions enable efficient handling of complex data by grouping related variables.
2. **Code Readability:** Using a structured approach reduces redundancy and enhances maintainability.
3. **Memory Management:** Unions optimize memory usage by sharing space between variables.
4. **Real-world Applications:** Both are widely used in applications such as databases, embedded systems, and networking.

Structures allow storing **multiple variables of different types** under the same name, whereas unions allow **different variables to share the same memory location**. Understanding these concepts is crucial for building efficient and modular programs.

CHAPTER 2: UNDERSTANDING STRUCTURES IN C

What is a Structure?

A **structure** in C is a user-defined data type that groups related variables of **different data types** together. It allows handling multiple attributes under a single unit.

Syntax of a Structure

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

Here, struct Student defines a structure with:

- name (character array of 50) to store student's name.
- age (integer) to store age.
- marks (float) to store marks.

Declaring and Accessing Structure Variables

```
#include <stdio.h>

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

```
};


int main() {
    struct Student s1 = {"Alice", 20, 85.5};


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


    return 0;
}
```

Output

Student Name: Alice

Student Age: 20

Student Marks: 85.50

This example shows how structures **group multiple variables together** and how to **initialize and access** them.

Advantages of Structures

1. **Data Grouping:** Allows storing different data types in a single variable.
2. **Improves Readability:** Makes programs easier to understand.
3. **Facilitates Code Modularity:** Enables efficient data handling.

Limitations of Structures

1. **Memory Usage:** Each member has its own memory space.
2. **Limited Memory Optimization:** Unlike unions, structures do not allow memory sharing.

CHAPTER 3: POINTERS TO STRUCTURES

Using Pointers with Structures

Pointers can be used to dynamically allocate memory for structures, making data manipulation more efficient.

Example: Using Pointers with Structures

```
#include <stdio.h>

#include <stdlib.h>


struct Student {

   char name[50];

   int age;

   float marks;

};


int main() {

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

```
    printf("Enter name: ");
    scanf("%s", s->name);
    printf("Enter age: ");
    scanf("%d", &s->age);
    printf("Enter marks: ");
    scanf("%f", &s->marks);

    printf("\nStudent Details:\n");
    printf("Name: %s\n", s->name);
    printf("Age: %d\n", s->age);
    printf("Marks: %.2f\n", s->marks);

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

Key Learnings

- -> is used to access structure members via a pointer.
- **Dynamic memory allocation** ensures memory is allocated efficiently.

CHAPTER 4: UNDERSTANDING UNIONS IN C

What is a Union?

A **union** is similar to a structure but with one key difference: **all members share the same memory location**. This makes unions **memory-efficient** when storing mutually exclusive data.

Syntax of a Union

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

Here, union Data can store **either** an integer, a float, or a string, but **not all at once**.

Example: Declaring and Accessing a Union

```
#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);

    return 0;
}
```

Output

Integer: 10

Float: 220.50

Since the same memory is used for i and f, changing f overwrites i.

Advantages of Unions

1. **Efficient Memory Usage:** Only one variable holds data at a time.
2. **Useful in Embedded Systems:** Ideal for hardware memory registers.
3. **Flexible Storage:** Best when handling different types of data at different times.

Disadvantages of Unions

1. **Overwrites Previous Data:** Only one member can hold data at any given time.
2. **Limited Use Cases:** Not suitable for storing multiple values simultaneously.

Chapter 5: Differences Between Structures and Unions

Feature	Structure	Union
Memory Usage	Allocates separate memory for each member.	Shares memory among all members.
Size	Sum of all members' sizes.	Size of the largest member.
Data Storage	All members can store values simultaneously.	Only one member stores a value at a time.
Use Cases	Ideal for grouping related data.	Useful for memory-constrained applications.

Chapter 6: Case Study – Employee Management System

Problem Statement

A company wants to store employee details. Some employees receive a **fixed salary**, while others work **hourly**. Using **structures and unions**, design an efficient program.

Solution

```
#include <stdio.h>

struct Employee {
   char name[50];
   int id;
   union {
      float fixed_salary;
      float hourly_wage;
   } pay;
   int isFixed; // 1 for fixed salary, 0 for hourly
};

int main() {
   struct Employee emp1 = {"John Doe", 101, .pay.fixed_salary = 5000, 1};
   struct Employee emp2 = {"Jane Smith", 102, .pay.hourly_wage = 25, 0};

   printf("Employee 1: %s, Salary: %.2f\n", emp1.name, emp1.isFixed ? emp1.pay.fixed_salary : 0);
   printf("Employee 2: %s, Hourly Wage: %.2f\n", emp2.name, emp2.isFixed ? 0 : emp2.pay.hourly_wage);
```

```
    return 0;
}
```

Key Learnings

- **Structures** store employee details.
- **Unions** manage salary types efficiently.
- **Saves memory** by storing either fixed salary or hourly wage, but not both.

CHAPTER 7: EXERCISES

Exercise 1: Implement a Student Database

Write a program to store student details using **structures**, including:

- Name
- Roll Number
- Marks in 3 subjects
- Average Marks Calculation

Exercise 2: Implement a Product Inventory System

Use **unions** to store:

- Product name
- Price
- Weight (for weighted items)
- Quantity (for counted items)

Exercise 3: Compare Memory Usage of Structures and Unions

Write a program to display the **memory size** occupied by a **structure and a union** with the same data members.

CONCLUSION

Structures and unions are powerful tools for organizing data in C. While structures allow storing multiple variables **simultaneously**, unions are more **memory-efficient** when handling different data types at different times. By mastering these concepts, programmers can **optimize memory usage and improve data handling** in real-world applications. 🚀

Nested Structures and Arrays of Structures in C

Chapter 1: Introduction to Structures in C

Structures in C are user-defined data types that allow the grouping of related variables under one name. Unlike arrays, which store multiple values of the same type, structures can store variables of different types together. Structures are widely used in real-world applications where complex data representation is required, such as managing student records, employee details, and database systems.

A structure is defined using the struct keyword, followed by the structure name and a block containing its members. These members can be of any valid C data type, including other structures, which leads to the concept of **nested structures**. Additionally, an **array of structures** allows multiple records of the same structure type to be managed efficiently. This combination provides flexibility in data storage and retrieval, making them a crucial concept in C programming.

Chapter 2: Nested Structures in C

Understanding Nested Structures

A **nested structure** is a structure that contains another structure as a member. This allows the grouping of related substructures within a larger structure, making data organization more efficient and meaningful. Nested structures are particularly useful when dealing with hierarchical data, such as a student's academic details within a student record or a department within an organization.

Syntax of Nested Structures

```
struct Address {
   char city[50];
   char state[50];
   int zip;
};

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

In this example, the Address structure is nested inside the Employee structure. Each Employee record will now contain not only an id and name but also an associated address.

Example: Using Nested Structures

```
#include <stdio.h>

struct Address {
   char city[50];
   char state[50];
   int zip;
};
```

```
struct Employee {
   char name[50];
   int id;
   struct Address address;
};

int main() {
   struct Employee emp1 = {"John Doe", 101, {"New York", "NY", 10001}};

   printf("Employee Name: %s\n", emp1.name);
   printf("Employee ID: %d\n", emp1.id);
   printf("City: %s, State: %s, ZIP: %d\n", emp1.address.city, emp1.address.state, emp1.address.zip);

   return 0;
}
```

This example demonstrates how a nested structure can be used to store and retrieve hierarchical data.

Exercise

1. Create a nested structure to store a book's details, including title, author, and publisher information.

2. Modify the employee structure to include multiple addresses (permanent and temporary) using another nested structure.
3. Implement a program to manage student academic records using nested structures.

CHAPTER 3: ARRAYS OF STRUCTURES IN C

Understanding Arrays of Structures

An **array of structures** is an array where each element is a structure. This is useful when dealing with multiple records of the same type, such as a list of students, employees, or products in a store. Instead of creating multiple structure variables separately, an array of structures enables handling them efficiently.

Syntax of Arrays of Structures

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

struct Student students[100]; // Declaring an array of structures
```

Here, students is an array of 100 Student structure elements, each containing name, roll_no, and marks.

Example: Using an Array of Structures

```
#include <stdio.h>

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

int main() {
    struct Student students[3] = {
        {"Alice", 101, 85.5},
        {"Bob", 102, 78.0},
        {"Charlie", 103, 92.3}
    };

    printf("Student Records:\n");
    for (int i = 0; i < 3; i++) {
        printf("Name: %s, Roll No: %d, Marks: %.2f\n",
               students[i].name, students[i].roll_no, students[i].marks);
    }
```

```
    return 0;
}
```

In this example, an array of structures stores details of multiple students and iterates through them to display the information.

Exercise

1. Create an array of structures to store details of five employees, including name, department, and salary.
2. Modify the previous program to accept input from the user instead of hardcoded values.
3. Implement a program to store and display the details of ten products in a store.

CHAPTER 4: COMBINING NESTED STRUCTURES WITH ARRAYS OF STRUCTURES

Using Nested Structures Inside an Array of Structures

We can also use **nested structures within an array of structures** to manage complex data efficiently. For example, an array of employees, where each employee has a nested address structure.

Example: Managing Multiple Employees with Nested Structures

```
#include <stdio.h>

struct Address {
    char city[50];
```

```
    char state[50];
    int zip;
};

struct Employee {
    char name[50];
    int id;
    struct Address address;
};

int main() {
    struct Employee employees[2] = {
        {"John Doe", 101, {"New York", "NY", 10001}},
        {"Jane Smith", 102, {"Los Angeles", "CA", 90001}}
    };

    printf("Employee Records:\n");
    for (int i = 0; i < 2; i++) {
        printf("Name: %s, ID: %d\n", employees[i].name,
employees[i].id);
        printf("City: %s, State: %s, ZIP: %d\n\n",
```

```
        employees[i].address.city, employees[i].address.state,
employees[i].address.zip);
    }


    return 0;
}
```

This example combines an array of structures with nested structures to store hierarchical data for multiple employees.

Exercise

1. Create an array of structures to manage ten students, where each student has a nested structure for subjects and marks.
2. Modify the previous program to include an input function that allows the user to enter details for multiple employees dynamically.
3. Implement a system to manage car rental services where each car has details about the owner, including nested structure information.

Case Study: Employee Management System Using Nested Structures and Arrays of Structures

A company wants to develop an **Employee Management System** to store information about its employees, including their personal details, job details, and address. The system must allow:

1. **Storing multiple employees' records using an array of structures.**

2. **Managing address information using nested structures.**
3. **Displaying the records in a formatted manner.**
4. **Allowing the user to add new employees dynamically.**

Implementation

- Define a structure Address for city, state, and ZIP code.
- Define a structure JobDetails for department and salary.
- Use an array of Employee structures, where each employee has Address and JobDetails as nested structures.
- Implement functions for adding, modifying, and displaying employee details.

By combining **nested structures** with **arrays of structures**, this system efficiently manages and organizes complex employee information.

CONCLUSION

- **Nested structures** allow hierarchical data representation.
- **Arrays of structures** help manage multiple records efficiently.
- **Combining both** enhances data organization for real-world applications.

File Handling in C (fopen, fclose, fread, fwrite)

Chapter 1: Introduction to File Handling in C

What is File Handling?

File handling in C allows programmers to store, retrieve, and manipulate data **persistently** rather than just during program execution. This ensures that data is saved even after the program terminates. C provides a set of **library functions** to perform file operations such as **reading, writing, and updating files**.

Why Use File Handling?

1. **Data Persistence** – Files store data permanently, unlike variables that lose data when the program exits.
2. **Large Data Handling** – Files manage large data efficiently compared to memory-based structures.
3. **Data Sharing** – Enables multiple programs to read and write shared data.
4. **Logging & Debugging** – Useful for maintaining logs and debugging applications.

Common File Handling Functions

Function	Description
fopen()	Opens a file in different modes (read, write, append).
fclose()	Closes an open file and releases resources.
fread()	Reads data from a file.

fwrite()	Writes data to a file.

File operations are performed using the **FILE pointer,** defined in the <stdio.h> library.

CHAPTER 2: OPENING AND CLOSING A FILE

Using fopen() to Open a File

The fopen() function is used to **open a file** in different modes.

Syntax

FILE *fopen(const char *filename, const char *mode);

- **filename** – Name of the file to open.
- **mode** – Specifies the operation (read, write, append).

Modes in fopen()

Mode	Description
"r"	Open for reading (file must exist).
"w"	Open for writing (creates a new file or erases an existing one).
"a"	Open for appending (adds data to an existing file).
"r+"	Open for reading and writing (file must exist).
"w+"	Open for reading and writing (erases existing file).
"a+"	Open for reading and appending.

Example: Opening a File

```
#include <stdio.h>


int main() {
   FILE *file = fopen("example.txt", "w"); // Open file in write mode
   if (file == NULL) {
      printf("Error opening file!\n");
      return 1;
   }
   printf("File opened successfully.\n");
   fclose(file); // Close the file
   return 0;
}
```

Output

File opened successfully.

Closing a File Using fclose()

The fclose() function **closes an open file** and releases system resources.

Syntax

int fclose(FILE *file);

- file – The file pointer to be closed.
- Returns **0 on success, EOF on failure**.

CHAPTER 3: WRITING TO A FILE USING FWRITE()

Using fwrite() to Store Data in a File

The fwrite() function writes **binary data** to a file.

Syntax

size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream);

- ptr – Pointer to the data.
- size – Size of each element.
- count – Number of elements.
- stream – The file pointer.

Example: Writing Data to a File

```
#include <stdio.h>


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


int main() {
   FILE *file = fopen("students.dat", "wb");
```

```
    struct Student s1 = {"Alice", 20, 85.5};


    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }


    fwrite(&s1, sizeof(struct Student), 1, file);
    printf("Data written successfully.\n");


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

Output

Data written successfully.

This program creates a **binary file (students.dat)** and stores student details.

CHAPTER 4: READING FROM A FILE USING FREAD()

Using fread() to Retrieve Data

The fread() function reads **binary data** from a file.

Syntax

size_t fread(void *ptr, size_t size, size_t count, FILE *stream);

- ptr – Pointer where data is stored.
- size – Size of each element.
- count – Number of elements.
- stream – The file pointer.

Example: Reading Data from a File

```
#include <stdio.h>

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

int main() {
   FILE *file = fopen("students.dat", "rb");
   struct Student s1;

   if (file == NULL) {
      printf("Error opening file!\n");
```

```
    return 1;
  }

  fread(&s1, sizeof(struct Student), 1, file);
  printf("Student Name: %s\n", s1.name);
  printf("Age: %d\n", s1.age);
  printf("Marks: %.2f\n", s1.marks);

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

Output

Student Name: Alice

Age: 20

Marks: 85.50

This program **reads** student details from students.dat and prints them.

CHAPTER 5: CASE STUDY – EMPLOYEE RECORD SYSTEM

Problem Statement

A company wants to **store, retrieve, and update** employee records efficiently.

Solution Using File Handling

Operations to Implement

1. Add an employee record (fwrite()).
2. Read all employee records (fread()).

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

struct Employee {
   char name[50];
   int id;
   float salary;
};

void addEmployee() {
   FILE *file = fopen("employees.dat", "ab");
   struct Employee emp;

   printf("Enter Name: ");
   scanf(" %[^\n]", emp.name);
   printf("Enter ID: ");
   scanf("%d", &emp.id);
```

```
    printf("Enter Salary: ");
    scanf("%f", &emp.salary);

    fwrite(&emp, sizeof(struct Employee), 1, file);
    fclose(file);
    printf("Employee record added successfully.\n");
}

void displayEmployees() {
    FILE *file = fopen("employees.dat", "rb");
    struct Employee emp;

    if (file == NULL) {
        printf("No employee records found.\n");
        return;
    }

    while (fread(&emp, sizeof(struct Employee), 1, file)) {
        printf("\nName: %s\nID: %d\nSalary: %.2f\n", emp.name,
emp.id, emp.salary);
    }
```

```
    fclose(file);
}

int main() {
    int choice;
    while (1) {
        printf("\n1. Add Employee\n2. View Employees\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: addEmployee(); break;
            case 2: displayEmployees(); break;
            case 3: exit(0);
            default: printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

Expected Output

1. Add Employee

2. View Employees

3. Exit

Enter choice: 1

Enter Name: John Doe

Enter ID: 101

Enter Salary: 5000

Employee record added successfully.

Enter choice: 2

Name: John Doe

ID: 101

Salary: 5000.00

Key Learnings

- **Appending data** using "ab" mode.
- **Reading multiple records** using fread() in a loop.

CHAPTER 6: EXERCISES

Exercise 1: Create a Student Management System

- Store **student records** using fwrite().
- Retrieve and **display all records** using fread().

Exercise 2: Implement a Logging System

- Write program logs to a **text file (log.txt)** using fprintf().

Exercise 3: Modify Employee Salaries

- Allow the user to **search and update** salaries in a file.

Conclusion

File handling in C enables **persistent data storage**, making programs **more practical and scalable**. By mastering fopen(), fclose(), fread(), and fwrite(), programmers can efficiently manage **data storage, retrieval, and modification** for real-world applications. 🚀

READING AND WRITING TO FILES IN C

CHAPTER 1: INTRODUCTION TO FILE HANDLING IN C

File handling in C is a crucial feature that allows data to be stored permanently on storage devices rather than just in memory during program execution. Unlike variables and arrays, which store data temporarily, files enable reading, writing, modifying, and deleting data in a structured way. File handling operations are widely used in applications such as data logging, database management, and configuration settings.

C provides a set of file-handling functions in the stdio.h library to manage files efficiently. The key operations include:

- **Opening a file (fopen())**: Establishes a connection between a program and a file.
- **Closing a file (fclose())**: Terminates the connection.
- **Reading from a file (fgetc(), fgets(), fscanf())**: Extracts data from a file.
- **Writing to a file (fputc(), fputs(), fprintf())**: Stores data in a file.
- **Appending data (a mode in fopen())**: Adds new data to an existing file without modifying previous content.

Understanding how to read from and write to files in C is essential for efficient data management. The following sections will explore file operations in depth with examples, exercises, and real-world case studies.

CHAPTER 2: WRITING TO FILES IN C

Understanding File Writing

Writing to a file involves storing information in a file so that it can be retrieved later. The fopen() function is used to open a file in write mode ("w") or append mode ("a"). When a file is opened in "w" mode, it **overwrites** existing content, while "a" mode appends new data without affecting the existing data.

Syntax for Opening and Writing to a File

```
FILE *filePointer;

filePointer = fopen("filename.txt", "w"); // Open in write mode
```

- If the file does not exist, it is created.
- If the file exists, it is overwritten.

Example: Writing Data to a File

```
#include <stdio.h>


int main() {
   FILE *file;
   file = fopen("data.txt", "w");  // Open file in write mode


   if (file == NULL) {
      printf("Error opening file!\n");
      return 1;
```

```
    }

    fprintf(file, "Hello, this is a test file.\n");
    fprintf(file, "Writing data to a file in C.\n");

    fclose(file);  // Close the file
    printf("Data written successfully.\n");

    return 0;
}
```

Explanation:

- The file data.txt is created and opened in write mode.
- The fprintf() function writes formatted text to the file.
- The file is then closed using fclose() to ensure data integrity.

Exercise

1. Write a program that stores student names and grades in a file.
2. Modify the program to append new student records instead of overwriting existing data.
3. Implement a file-writing system that logs user activities in an application.

CHAPTER 3: READING FROM FILES IN C

Understanding File Reading

Reading from a file means extracting stored information and displaying it in a program. This is done using functions such as:

- fgetc(): Reads a single character from a file.
- fgets(): Reads a string from a file.
- fscanf(): Reads formatted input from a file.

Syntax for Opening and Reading a File

```
FILE *filePointer;

filePointer = fopen("filename.txt", "r"); // Open in read mode
```

- If the file exists, it is opened for reading.
- If the file does not exist, fopen() returns NULL.

Example: Reading Data from a File

```
#include <stdio.h>


int main() {
   FILE *file;
   char buffer[100];


   file = fopen("data.txt", "r");  // Open file in read mode


   if (file == NULL) {
```

```
        printf("Error opening file!\n");
        return 1;
    }

    printf("File content:\n");
    while (fgets(buffer, 100, file) != NULL) {
        printf("%s", buffer);  // Print each line of the file
    }

    fclose(file);  // Close the file
    return 0;
}
```

Explanation:

- The fgets() function reads one line at a time from the file.
- The loop continues until the end of the file (EOF) is reached.
- The file is then closed to free system resources.

Exercise

1. Create a program that reads and displays student records from a file.
2. Modify the program to count the number of lines in a file.
3. Implement a file reader that extracts specific information based on user input.

CHAPTER 4: READING AND WRITING STRUCTURES TO FILES

Using Structures for File Operations

C allows **storing and retrieving structured data** from files using fwrite() and fread(). This is useful for maintaining databases of employees, students, or products.

Example: Writing Structures to a File

```
#include <stdio.h>

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

int main() {
    struct Student s1 = {"Alice", 101, 89.5};
    FILE *file = fopen("students.dat", "wb");  // Open binary file

    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
```

```
    }

    fwrite(&s1, sizeof(struct Student), 1, file);
    fclose(file);

    printf("Student record saved successfully.\n");
    return 0;
}
```

Example: Reading Structures from a File

```
#include <stdio.h>

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

int main() {
    struct Student s1;
    FILE *file = fopen("students.dat", "rb");  // Open file in read mode
```

```
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fread(&s1, sizeof(struct Student), 1, file);
    fclose(file);

    printf("Student Details:\nName: %s\nRoll No: %d\nMarks: %.2f\n",
           s1.name, s1.roll_no, s1.marks);

    return 0;
}
```

Exercise

1. Modify the above program to store multiple student records.
2. Implement a search feature to retrieve a specific record.
3. Extend the program to update or delete existing records.

Case Study: Student Management System Using File Handling

A university wants to develop a **Student Management System** that:

1. **Stores student records in a file.**

2. **Allows adding, reading, and updating records.**
3. **Retrieves specific records based on roll number.**
4. **Maintains data persistence even after the program exits.**

Implementation Approach

1. Use **structures** to store student details (name, roll number, marks).
2. Implement **file writing (fwrite())** to store records.
3. Implement **file reading (fread())** to retrieve records.
4. Provide options for **searching, modifying, and deleting records**.

By implementing file handling techniques, the system ensures **efficient data storage, retrieval, and modification**.

CONCLUSION

- **Writing to files** enables permanent storage of data.
- **Reading from files** allows retrieval of stored information.
- **Using structures with files** enhances data management in real-world applications.
- **File operations** are essential for **database systems, log management, and persistent storage**.

ERROR HANDLING IN FILE OPERATIONS IN C

CHAPTER 1: INTRODUCTION TO ERROR HANDLING IN FILE OPERATIONS

What is Error Handling in File Operations?

Error handling in file operations ensures that a program **handles unexpected errors** such as missing files, insufficient permissions, or disk failures gracefully. Proper error handling **prevents crashes** and improves the reliability of applications.

Why is Error Handling Important?

1. **Prevents Program Crashes** – Ensures smooth execution even if file operations fail.
2. **Enhances User Experience** – Provides meaningful error messages instead of abrupt termination.
3. **Ensures Data Integrity** – Prevents accidental data loss or corruption.
4. **Handles Different System Environments** – Deals with issues like missing files or disk read/write restrictions.

Common Errors in File Handling

Error Type	Possible Cause
File Not Found	The file does not exist at the specified location.
Permission Denied	Insufficient privileges to access the file.
Disk Full	No space left on the device for writing.

File Already Exists	Attempting to create a file that already exists (with "w" mode).
Read/Write Error	Hardware failure or corruption in the file.

To handle these errors, we use **error-checking techniques** and appropriate C library functions.

CHAPTER 2: USING FOPEN() WITH ERROR HANDLING

Checking for File Opening Errors

The fopen() function returns **NULL** if the file cannot be opened. We must **check for NULL** before proceeding.

Syntax of fopen()

FILE *fopen(const char *filename, const char *mode);

- Returns a **FILE pointer** if successful.
- Returns **NULL** if an error occurs.

Example: Checking for File Opening Errors

```
#include <stdio.h>


int main() {
   FILE *file = fopen("nonexistent.txt", "r");


   if (file == NULL) {
```

```
    perror("Error opening file"); // Prints system-defined error
message
    return 1; // Exit with error code
  }

  printf("File opened successfully.\n");
  fclose(file);

  return 0;
}
```

Possible Output

Error opening file: No such file or directory

Key Learnings

- **Using perror()** – Displays system-generated error messages.
- **Checking NULL return from fopen()** – Prevents segmentation faults.
- **Returning 1 to indicate failure** – Helps track program execution status.

CHAPTER 3: HANDLING FILE CLOSING ERRORS USING FCLOSE()

Why Check for fclose() Errors?

- fclose() **may fail** if the file was already closed or there is a disk error.
- It returns **EOF (-1)** on failure.

Example: Checking for fclose() Errors

```
#include <stdio.h>


int main() {
   FILE *file = fopen("example.txt", "w");
   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }

   if (fclose(file) == EOF) {
      perror("Error closing file");
      return 1;
   }

   printf("File closed successfully.\n");
   return 0;
}
```

Expected Output

File closed successfully.

If an error occurs, perror("Error closing file") will display the error.

CHAPTER 4: HANDLING READ/WRITE ERRORS (FREAD() AND FWRITE())

Using fread() with Error Checking

The fread() function reads **binary data** from a file. It returns the **number of elements successfully read**.

Example: Checking Read Errors

```
#include <stdio.h>


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


int main() {
   FILE *file = fopen("students.dat", "rb");
   struct Student s1;
```

```
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    if (fread(&s1, sizeof(struct Student), 1, file) != 1) {
        perror("Error reading file");
        fclose(file);
        return 1;
    }

    printf("Student Name: %s\n", s1.name);
    fclose(file);
    return 0;
}
```

Key Takeaways

- **Checking fread() return value** ensures all data is read correctly.
- **If fread() returns 0,** an error or end-of-file (EOF) is encountered.

Using fwrite() with Error Checking

fwrite() writes **binary data** to a file. It returns the **number of elements written**.

```
#include <stdio.h>


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


int main() {
   FILE *file = fopen("students.dat", "wb");
   struct Student s1 = {"Alice", 20, 85.5};


   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }


   if (fwrite(&s1, sizeof(struct Student), 1, file) != 1) {
      perror("Error writing file");
```

```
    fclose(file);
    return 1;
  }

  printf("Data written successfully.\n");
  fclose(file);
  return 0;
}
```

Key Takeaways

- Always **check if fwrite() successfully writes data**.
- Handling **disk full errors** prevents partial writes.

CHAPTER 5: HANDLING END-OF-FILE AND UNEXPECTED ERRORS

Checking for EOF While Reading

The function **feof(FILE *file)** detects **end-of-file (EOF)** during reading.

```
#include <stdio.h>

int main() {
  FILE *file = fopen("students.dat", "rb");
  if (file == NULL) {
```

```
    perror("Error opening file");
    return 1;
  }

  while (!feof(file)) {
    char ch = fgetc(file);
    if (feof(file))
      break;
    putchar(ch);
  }

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

Handling Other Errors

Function	Purpose
perror("Error")	Prints system error message.
clearerr(file)	Clears file error flags.
ferror(file)	Checks if an error occurred.

Example:

```
if (ferror(file)) {
```

```
    printf("File error occurred.\n");
}
```

CHAPTER 6: CASE STUDY – SECURE FILE HANDLING IN A BANKING SYSTEM

Problem Statement

A bank system needs to store **account details** securely. It must:

1. **Handle missing files.**
2. **Prevent data corruption.**
3. **Verify all read/write operations.**

Solution Using Robust Error Handling

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

struct Account {
    int accountNumber;
    float balance;
};

void createAccount() {
    FILE *file = fopen("accounts.dat", "wb");
```

```
    if (file == NULL) {
        perror("Error creating account file");
        exit(1);
    }

    struct Account acc = {1001, 5000.00};
    if (fwrite(&acc, sizeof(struct Account), 1, file) != 1) {
        perror("Error writing account data");
        fclose(file);
        exit(1);
    }

    printf("Account created successfully.\n");
    fclose(file);
}

void viewAccount() {
    FILE *file = fopen("accounts.dat", "rb");
    if (file == NULL) {
        perror("Error opening account file");
        exit(1);
```

```
    }

    struct Account acc;
    if (fread(&acc, sizeof(struct Account), 1, file) != 1) {
        perror("Error reading account data");
        fclose(file);
        exit(1);
    }

    printf("Account Number: %d\nBalance: %.2f\n",
acc.accountNumber, acc.balance);
    fclose(file);
}

int main() {
    createAccount();
    viewAccount();
    return 0;
}
```

Key Learnings

- **Verifies every file operation** to prevent crashes.

- **Prevents partial file corruption** with exit(1).

CHAPTER 7: EXERCISES

1. **Modify the above case study** to allow users to update their balance.
2. **Implement a log file system** that stores file access errors.
3. **Create a student database** with robust error handling.

CONCLUSION

Error handling in file operations is essential for building **robust and secure applications**. By implementing proper **file opening, reading, writing, and EOF detection,** programmers can prevent **data loss and crashes,** making applications **more reliable**. 🚀

UNDERSTANDING TYPEDEF AND ENUM IN C

CHAPTER 1: INTRODUCTION TO TYPEDEF AND ENUM IN C

C provides several features to enhance code readability and maintainability. Among them, **typedef** and **enum** play a crucial role in defining custom data types and handling named constants efficiently.

- **typedef (Type Definition)**: Used to create custom names (aliases) for existing data types. This improves code readability, simplifies complex declarations, and makes code more maintainable.
- **enum (Enumeration)**: Used to define a list of named integer constants, making programs more understandable and reducing errors associated with magic numbers.

Both typedef and enum are widely used in **large-scale applications** where code clarity, reusability, and maintainability are essential. This chapter explores these concepts in depth with examples, exercises, and real-world case studies.

CHAPTER 2: UNDERSTANDING TYPEDEF IN C

What is typedef?

The typedef keyword in C is used to create an alias (alternative name) for an existing data type. This helps improve code clarity, especially when working with complex structures or pointers.

Syntax of typedef

```
typedef existing_type new_name;
```

For example:

typedef unsigned int uint;

uint age = 25; // Equivalent to unsigned int age = 25;

Example: Using typedef for Data Types

```
#include <stdio.h>


typedef unsigned long int ULong;


int main() {
   ULong population = 7800000000;  // Equivalent to unsigned long int
   printf("World Population: %lu\n", population);
   return 0;
}
```

Using typedef with Structures

When dealing with structures, typedef simplifies the declaration process.

```
#include <stdio.h>


// Define structure using typedef
typedef struct {
   char name[50];
```

```
    int age;
    float salary;
} Employee;

int main() {
    Employee emp1 = {"John Doe", 30, 50000.50};

    printf("Name: %s, Age: %d, Salary: %.2f\n", emp1.name, emp1.age, emp1.salary);
    return 0;
}
```

Why Use typedef with Structures?

- Avoids writing struct keyword repeatedly.
- Improves code readability.
- Useful when working with **arrays of structures**.

Exercise

1. Define a typedef for float as RealNumber and use it in a program.
2. Create a typedef for struct Book containing title, author, and price.
3. Implement a typedef for a function pointer and use it in a simple program.

CHAPTER 3: UNDERSTANDING ENUM IN C

What is an Enumeration (enum)?

An **enumeration** is a user-defined type that assigns names to a set of integer constants. This makes the program more readable, as it replaces magic numbers with meaningful names.

Syntax of enum

enum EnumName {constant1, constant2, constant3, ...};

By default:

- The first constant is assigned 0, the second 1, and so on.
- You can also manually assign values.

Example: Using enum for Days of the Week

```
#include <stdio.h>


enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY};


int main() {
   enum Day today = WEDNESDAY;


   printf("Today is day number %d of the week.\n", today);  //
Outputs 3

   return 0;
```

```
}
```

Advantages of enum:

- Increases code readability.
- Prevents accidental use of incorrect numbers.
- Groups related constants together.

Using enum with Explicit Values

```
#include <stdio.h>

enum Status {SUCCESS = 1, FAILURE = 0, PENDING = 2};

int main() {
   enum Status orderStatus = SUCCESS;

   if (orderStatus == SUCCESS) {
      printf("Order was successful!\n");
   } else {
      printf("Order failed or is pending.\n");
   }

   return 0;
}
```

Here, assigning values manually allows us to represent meaningful states using readable names.

Exercise

1. Define an enum for months of the year and print the corresponding integer values.
2. Modify the above orderStatus program to check for multiple statuses.
3. Create an enum to represent traffic light signals (RED, YELLOW, GREEN) and implement a program that prints their meaning.

CHAPTER 4: COMBINING TYPEDEF AND ENUM IN C

Using typedef with enum

The typedef keyword can be used with enum to create **shorter and cleaner** type definitions.

Example: Defining an Alias for an Enumeration

```
#include <stdio.h>


// Defining a typedef for enum

typedef enum {RED, YELLOW, GREEN} TrafficLight;


int main() {

   TrafficLight signal = GREEN;
```

```
    if (signal == GREEN) {
        printf("Go!\n");
    } else if (signal == YELLOW) {
        printf("Slow down!\n");
    } else {
        printf("Stop!\n");
    }

    return 0;
}
```

Here, using typedef removes the need to write enum each time.

Using typedef with enum and Structures

```
#include <stdio.h>

typedef enum {ADMIN, MANAGER, EMPLOYEE} Role;

// Define structure with typedef
typedef struct {
    char name[50];
    int id;
```

```
    Role role;
} Person;


int main() {
    Person p1 = {"Alice", 101, MANAGER};

    printf("Name: %s, ID: %d, Role: %d\n", p1.name, p1.id, p1.role);
    return 0;
}
```

In this example:

- typedef is used with enum for better readability.
- typedef is used with struct to avoid repetitive struct keyword usage.

Exercise

1. Define an enum for vehicle types and use typedef to create an alias.
2. Modify the traffic signal program to accept user input and print the corresponding action.
3. Implement a structure typedef for a **movie database**, using enum to categorize genres.

Case Study: Employee Management System Using typedef and enum

A company needs an **Employee Management System** where:

1. Employees have **unique IDs, names, and roles** (Manager, Developer, HR).
2. The system must ensure role categorization using enum.
3. The system should use typedef to define structures efficiently.

Implementation Plan

- Define an **enum Role** (MANAGER, DEVELOPER, HR).
- Create a **typedef struct Employee** with:
  - char name[50]
  - int id
  - Role role
- Use an **array of structures** to store multiple employees.
- Implement a function to **display employee details**.

Example Implementation

```
#include <stdio.h>


typedef enum {MANAGER, DEVELOPER, HR} Role;


typedef struct {
  char name[50];
  int id;
```

```
    Role role;
} Employee;

void displayEmployee(Employee e) {
    printf("ID: %d, Name: %s, Role: ", e.id, e.name);
    switch (e.role) {
        case MANAGER: printf("Manager\n"); break;
        case DEVELOPER: printf("Developer\n"); break;
        case HR: printf("HR\n"); break;
        default: printf("Unknown\n");
    }
}

int main() {
    Employee emp1 = {"John Doe", 101, MANAGER};
    Employee emp2 = {"Alice Smith", 102, DEVELOPER};

    displayEmployee(emp1);
    displayEmployee(emp2);

    return 0;
```

```
}
```

Benefits of Using typedef and enum Here:

- **Enum ensures role consistency** (MANAGER, DEVELOPER, HR).
- **Typedef simplifies structure usage**.
- **Improved readability and maintainability**.

CONCLUSION

- **typedef** enhances readability and simplifies complex type definitions.
- **enum** improves code clarity by replacing magic numbers with named constants.
- **Combining typedef and enum** makes code more structured and manageable.
- **Real-world applications** include database systems, embedded systems, and role-based access control.

COMMAND-LINE ARGUMENTS IN C

CHAPTER 1: INTRODUCTION TO COMMAND-LINE ARGUMENTS

What Are Command-Line Arguments?

Command-line arguments in C allow users to **pass input values** to a program while executing it in the terminal. Instead of taking input through scanf() or fgets(), command-line arguments **provide input directly from the command line** when running the program.

Why Use Command-Line Arguments?

1. **Automate Input Processing** – Eliminates the need for manual input during execution.
2. **Enhance Script Flexibility** – Enables passing different inputs without modifying code.
3. **Enable Batch Processing** – Useful in running multiple commands in a script.
4. **Improve Program Efficiency** – Saves time by avoiding interactive input.

Example Usage

If a program calculator is executed as:

./calculator 5 10

Here, 5 and 10 are **command-line arguments**.

CHAPTER 2: UNDERSTANDING ARGC AND ARGV

Command-line arguments are handled using two parameters in the main() function:

Syntax

int main(int argc, char *argv[])

- **argc (Argument Count)** – Holds the total number of command-line arguments (including the program name).
- **argv (Argument Vector)** – An array of character pointers storing each argument as a string.

Example: Displaying Command-Line Arguments

```
#include <stdio.h>


int main(int argc, char *argv[]) {
   printf("Number of arguments: %d\n", argc);

   for (int i = 0; i < argc; i++) {
      printf("Argument %d: %s\n", i, argv[i]);
   }

   return 0;
}
```

Output

$ ./program Hello World

Number of arguments: 3

Argument 0: ./program

Argument 1: Hello

Argument 2: World

Key Takeaways

- argv[0] always holds the program name (./program).
- Arguments are stored as **strings**, even if they represent numbers.

CHAPTER 3: CONVERTING ARGUMENTS FROM STRING TO INTEGER

Since argv[] stores arguments as **strings**, numerical inputs must be converted using atoi() (ASCII to Integer) or atof() (ASCII to Float).

Example: Adding Two Numbers from Command-Line Arguments

```
#include <stdio.h>

#include <stdlib.h>


int main(int argc, char *argv[]) {

  if (argc != 3) {

    printf("Usage: %s num1 num2\n", argv[0]);

    return 1;

  }
```

```
    int num1 = atoi(argv[1]);

    int num2 = atoi(argv[2]);

    printf("Sum: %d\n", num1 + num2);


    return 0;
}
```

Execution & Output

```
$ ./sum 5 10

Sum: 15
```

Key Learnings

- **atoi(argv[1]) converts a string to an integer**.
- The program validates argc to **ensure two numbers are provided**.

CHAPTER 4: HANDLING ERRORS IN COMMAND-LINE ARGUMENTS

Command-line arguments must be properly validated to prevent unexpected behavior.

Example: Handling Invalid Input

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>
```

```
int isNumber(char *str) {
   for (int i = 0; str[i] != '\0'; i++) {
      if (!isdigit(str[i])) return 0; // Return false if non-numeric character found
   }
   return 1;
}

int main(int argc, char *argv[]) {
   if (argc != 3) {
      printf("Usage: %s num1 num2\n", argv[0]);
      return 1;
   }

   if (!isNumber(argv[1]) || !isNumber(argv[2])) {
      printf("Error: Both arguments must be integers.\n");
      return 1;
   }

   int num1 = atoi(argv[1]);
   int num2 = atoi(argv[2]);
```

```
    printf("Product: %d\n", num1 * num2);


    return 0;
}
```

Execution & Output

Valid Input

$ ./multiply 5 6

Product: 30

Invalid Input

$ ./multiply 5 abc

Error: Both arguments must be integers.

Key Takeaways

- **Checking input validity using isdigit()** prevents runtime errors.
- **Error messages** guide the user on correct usage.

CHAPTER 5: USING COMMAND-LINE ARGUMENTS IN FILE OPERATIONS

Command-line arguments can be used to **read and write files dynamically,** allowing flexible file handling.

Example: Copying a File Using Command-Line Arguments

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

```
#include <stdlib.h>


int main(int argc, char *argv[]) {
   if (argc != 3) {
      printf("Usage: %s source_file destination_file\n", argv[0]);
      return 1;
   }

   FILE *source = fopen(argv[1], "r");
   if (source == NULL) {
      perror("Error opening source file");
      return 1;
   }

   FILE *destination = fopen(argv[2], "w");
   if (destination == NULL) {
      perror("Error creating destination file");
      fclose(source);
      return 1;
   }
```

```
    char ch;
    while ((ch = fgetc(source)) != EOF) {
        fputc(ch, destination);
    }

    printf("File copied successfully.\n");

    fclose(source);
    fclose(destination);
    return 0;
}
```

Execution & Output

```
$ ./copy source.txt destination.txt
File copied successfully.
```

Key Takeaways

- argv[1] specifies the **source file**.
- argv[2] specifies the **destination file**.
- The program validates **file existence and permissions**.

CHAPTER 6: CASE STUDY – COMMAND-LINE CALCULATOR

Problem Statement

Develop a command-line calculator that **accepts an operator and two operands** and performs the corresponding operation.

Solution

```
#include <stdio.h>

#include <stdlib.h>


int main(int argc, char *argv[]) {
  if (argc != 4) {
    printf("Usage: %s num1 operator num2\n", argv[0]);
    return 1;
  }

  double num1 = atof(argv[1]);
  double num2 = atof(argv[3]);
  char operator = argv[2][0];

  switch (operator) {
    case '+': printf("Result: %.2f\n", num1 + num2); break;
    case '-': printf("Result: %.2f\n", num1 - num2); break;
    case '*': printf("Result: %.2f\n", num1 * num2); break;
    case '/':
```

```
        if (num2 == 0) {
            printf("Error: Division by zero.\n");
            return 1;
        }
        printf("Result: %.2f\n", num1 / num2);
        break;
    default:
        printf("Error: Unsupported operator.\n");
        return 1;
  }

  return 0;
}
```

Execution & Output

Valid Calculation

$ ./calculator 10 + 5

Result: 15.00

Error Handling

$ ./calculator 10 / 0

Error: Division by zero.

Key Takeaways

- **Uses atof() to handle floating-point numbers**.
- **Ensures valid operations using a switch statement**.

CHAPTER 7: EXERCISES

Exercise 1: String Manipulation

Write a C program that **accepts a string as a command-line argument** and **reverses** it.

Exercise 2: Find Maximum in a List

Write a program that **takes N numbers as command-line arguments** and finds the **maximum value**.

Exercise 3: Convert Text Case

Write a program that **reads a file and converts text to uppercase** using command-line arguments.

CONCLUSION

Command-line arguments in C provide a **flexible way to handle input dynamically**. By using argc and argv[], programmers can create **efficient, automated, and interactive applications** that take input directly from the command line. 🚀

MEMORY MANAGEMENT TECHNIQUES AND BEST PRACTICES IN C

CHAPTER 1: INTRODUCTION TO MEMORY MANAGEMENT

Memory management is a critical aspect of programming in C, as C does not provide automatic garbage collection like modern languages (e.g., Java or Python). Efficient memory management ensures that a program uses system memory effectively, avoids memory leaks, and prevents undefined behavior such as segmentation faults.

In C, memory is divided into the following sections:

1. **Stack**: Used for function calls and local variables.
2. **Heap**: Used for dynamic memory allocation (malloc, calloc, realloc, free).
3. **Data Segment**: Stores global and static variables.
4. **Code Segment**: Contains the compiled program instructions.

Effective memory management techniques include proper allocation, deallocation, and avoiding issues such as **memory leaks**, **dangling pointers**, and **buffer overflows**. Understanding these principles helps in developing efficient and bug-free applications.

CHAPTER 2: STATIC VS DYNAMIC MEMORY ALLOCATION

Static Memory Allocation

Static memory allocation refers to memory that is allocated **at compile time**. It includes:

- Global variables
- Static variables
- Fixed-size arrays

The memory remains allocated until the program terminates.

Example: Static Memory Allocation

```
#include <stdio.h>


void display() {
   static int count = 0;  // Static variable
   count++;
   printf("Function called %d times\n", count);
}


int main() {
   display();
   display();
   return 0;
}
```

Here, the count variable retains its value across function calls due to static memory allocation.

Dynamic Memory Allocation

Dynamic memory allocation allows memory to be allocated **at runtime** using functions like malloc(), calloc(), realloc(), and free(). The dynamically allocated memory is stored in the **heap**.

Example: Dynamic Memory Allocation

```
#include <stdio.h>

#include <stdlib.h>


int main() {

  int *ptr = (int*) malloc(5 * sizeof(int));  // Allocating memory for 5 integers


  if (ptr == NULL) {

    printf("Memory allocation failed!\n");

    return 1;

  }


  for (int i = 0; i < 5; i++) {

    ptr[i] = i + 1;

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

  }


  free(ptr);  // Free allocated memory
```

```
    return 0;
}
```

Key Takeaways:

- **Use malloc(), calloc(), and realloc()** for dynamic allocation.
- **Always use free()** to deallocate memory.
- **Check if memory allocation is successful** (NULL check).

Exercise

1. Implement a program to store student grades dynamically using malloc().
2. Modify the above program to use calloc() instead of malloc().
3. Implement a dynamic array that grows in size using realloc().

CHAPTER 3: COMMON MEMORY MANAGEMENT ISSUES

Memory Leaks

A **memory leak** occurs when allocated memory is **not freed**. Over time, this reduces available memory, leading to performance degradation.

Example of Memory Leak

```
#include <stdlib.h>


void allocateMemory() {
    int *ptr = (int*) malloc(100 * sizeof(int));
```

```
    // Memory allocated but not freed
}

int main() {
    allocateMemory();
    return 0;
}
```

Here, memory is allocated but never freed, causing a memory leak.

Fixing Memory Leaks

- **Use free(ptr)** after dynamic allocation.
- **Set pointers to NULL** after freeing memory.
- **Use tools like Valgrind** to detect memory leaks.

Dangling Pointers

A **dangling pointer** occurs when a pointer still refers to memory that has been freed.

Example of a Dangling Pointer

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

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

```
    *ptr = 10;
    free(ptr);  // Memory freed
    printf("%d", *ptr);  // Undefined behavior
    return 0;
}
```

Fixing Dangling Pointers

- After free(), **set the pointer to NULL**.
- Avoid accessing memory after freeing it.

Exercise

1. Identify the memory leak in a given C program and fix it.
2. Implement a function that allocates and frees memory correctly.
3. Write a program that detects and prevents dangling pointers.

CHAPTER 4: BEST PRACTICES FOR MEMORY MANAGEMENT

1. Always Free Allocated Memory

Every call to malloc(), calloc(), or realloc() should have a corresponding free().

Example

```
int *ptr = (int*) malloc(10 * sizeof(int));
free(ptr);  // Always free allocated memory
```

2. Use calloc() Instead of malloc() for Zero Initialization

calloc() initializes memory to zero, which prevents undefined values.

Example

```
int *ptr = (int*) calloc(10, sizeof(int));  // Initializes all elements to 0
```

3. Check for NULL After Memory Allocation

Always check if memory allocation is successful.

Example

```
int *ptr = (int*) malloc(10 * sizeof(int));
if (ptr == NULL) {
   printf("Memory allocation failed!\n");
   return 1;
}
```

4. Avoid Memory Overflows

Buffer overflows occur when memory is accessed beyond allocated limits.

Example of Buffer Overflow

```
int arr[5];
arr[10] = 100;  // Accessing out of bounds (undefined behavior)
```

5. Use Smart Pointers in C++

For C++, prefer **smart pointers (std::unique_ptr, std::shared_ptr)** to manage memory automatically.

Case Study: Memory Management in a Banking System

A bank wants to develop a **Customer Management System** that:

1. Dynamically allocates memory for customer details.
2. Prevents **memory leaks and dangling pointers**.
3. Ensures memory is freed after a customer leaves.

Implementation

- Define a struct Customer containing name, account number, and balance.
- Use malloc() to allocate memory dynamically.
- Use realloc() to resize memory when customers increase.
- Ensure free() is used when a customer leaves.

Example Code

```
#include <stdio.h>

#include <stdlib.h>


typedef struct {

  char name[50];

  int account_no;

  float balance;

} Customer;


int main() {
```

```
Customer *customers;
int count = 2;

customers = (Customer*) malloc(count * sizeof(Customer));
if (customers == NULL) {
    printf("Memory allocation failed\n");
    return 1;
}

for (int i = 0; i < count; i++) {
    printf("Enter Name, Account No, and Balance: ");
    scanf("%s %d %f", customers[i].name,
&customers[i].account_no, &customers[i].balance);
}

for (int i = 0; i < count; i++) {
    printf("Customer: %s, Account No: %d, Balance: %.2f\n",
        customers[i].name, customers[i].account_no,
customers[i].balance);
}

free(customers);  // Free allocated memory
```

```
    return 0;
}
```

Key Takeaways

- **Efficient memory management prevents system crashes.**
- **Dynamic allocation ensures scalability.**
- **Proper deallocation avoids memory leaks.**

CONCLUSION

- **Memory management is crucial for efficient program execution.**
- **Static memory is fixed, while dynamic memory provides flexibility.**
- **Common issues include memory leaks, dangling pointers, and buffer overflows.**
- **Following best practices ensures optimized and secure applications.**

HANDS-ON CODING PRACTICE IN C

CHAPTER 1: INTRODUCTION TO HANDS-ON CODING IN C

Why is Hands-on Coding Important?

Hands-on coding is a crucial aspect of mastering **C programming**. Unlike theoretical learning, practicing coding helps to:

1. **Develop Problem-Solving Skills** – Improves logical thinking and debugging techniques.
2. **Strengthen Fundamentals** – Reinforces concepts like loops, functions, pointers, and memory management.
3. **Enhance Efficiency** – Teaches how to write optimized and error-free code.
4. **Gain Confidence** – Increases familiarity with real-world programming challenges.

In this chapter, we will explore various **hands-on exercises** ranging from **basic to advanced levels** to help build a strong foundation in C.

CHAPTER 2: BASIC HANDS-ON CODING EXERCISES

Exercise 1: Print "Hello, World!"

This is the first program every C programmer writes.

Code Implementation

```
#include <stdio.h>


int main() {
```

```
    printf("Hello, World!\n");

    return 0;

}
```

Expected Output

Hello, World!

Key Learnings

- **Syntax of C**: Understanding #include, main(), and printf().
- **Compiling and Running**: Using gcc program.c -o program and ./program.

Exercise 2: Simple Calculator

Objective: Implement a simple calculator that performs addition, subtraction, multiplication, and division.

Code Implementation

```
#include <stdio.h>


int main() {

    float num1, num2;

    char operator;


    printf("Enter an operator (+, -, *, /): ");

    scanf(" %c", &operator);
```

```
    printf("Enter two numbers: ");
    scanf("%f %f", &num1, &num2);

    switch (operator) {
        case '+': printf("Result: %.2f\n", num1 + num2); break;
        case '-': printf("Result: %.2f\n", num1 - num2); break;
        case '*': printf("Result: %.2f\n", num1 * num2); break;
        case '/':
            if (num2 == 0) {
                printf("Error: Division by zero is not allowed.\n");
            } else {
                printf("Result: %.2f\n", num1 / num2);
            }
            break;
        default:
            printf("Error: Invalid operator.\n");
    }

    return 0;
}
```

Expected Output

Enter an operator (+, -, *, /): *

Enter two numbers: 4 5

Result: 20.00

Key Learnings

- **Switch statements** for decision-making.
- **Handling division by zero** to prevent errors.

CHAPTER 3: INTERMEDIATE HANDS-ON CODING

Exercise 3: Find the Largest Number in an Array

Objective: Write a program to find the largest number in an array using pointer arithmetic.

Code Implementation

```
#include <stdio.h>


int findLargest(int *arr, int size) {
   int *ptr = arr;
   int largest = *ptr;


   for (int i = 1; i < size; i++) {
      if (*(ptr + i) > largest) {
```

```
            largest = *(ptr + i);
        }
    }
    return largest;
}

int main() {
    int arr[] = {10, 25, 38, 42, 5};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Largest number: %d\n", findLargest(arr, size));

    return 0;
}
```

Expected Output

Largest number: 42

Key Learnings

- **Using pointers for array traversal**.
- **Efficient memory access with pointer arithmetic**.

Exercise 4: Reverse a String Using Pointers

Objective: Reverse a given string using pointers.

Code Implementation

```
#include <stdio.h>

#include <string.h>


void reverseString(char *str) {
   int len = strlen(str);
   char *left = str, *right = str + len - 1;

   while (left < right) {
      char temp = *left;
      *left = *right;
      *right = temp;
      left++;
      right--;
   }
}

int main() {
   char str[100];
```

```
    printf("Enter a string: ");
    scanf("%s", str);

    reverseString(str);
    printf("Reversed string: %s\n", str);

    return 0;
}
```

Expected Output

Enter a string: hello

Reversed string: olleh

Key Learnings

- **Using pointers to manipulate character arrays**.
- **Efficient swapping without additional memory**.

CHAPTER 4: ADVANCED HANDS-ON CODING

Exercise 5: Implement Dynamic Memory Allocation for Student Records

Objective: Store student records dynamically and display them.

Code Implementation

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

```
#include <stdlib.h>

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

int main() {
    int n;
    printf("Enter number of students: ");
    scanf("%d", &n);

    struct Student *students = (struct Student*)malloc(n *
sizeof(struct Student));

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

```
    for (int i = 0; i < n; i++) {
        printf("Enter details for Student %d:\n", i + 1);
        printf("Name: ");
        scanf(" %[^\n]", students[i].name);
        printf("Age: ");
        scanf("%d", &students[i].age);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }

    printf("\nStudent Records:\n");
    for (int i = 0; i < n; i++) {
        printf("Student %d: %s, Age: %d, Marks: %.2f\n", i + 1,
students[i].name, students[i].age, students[i].marks);
    }

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

Expected Output

Enter number of students: 2

Enter details for Student 1:

Name: Alice

Age: 20

Marks: 85.5

Enter details for Student 2:

Name: Bob

Age: 22

Marks: 90.0

Student Records:

Student 1: Alice, Age: 20, Marks: 85.50

Student 2: Bob, Age: 22, Marks: 90.00

Key Learnings

- **Dynamic memory allocation using malloc()**.
- **Efficiently handling multiple records**.
- **Freeing memory with free() to prevent memory leaks**.

CHAPTER 5: CASE STUDY – COMMAND-LINE FILE HANDLING

Problem Statement

Create a program that reads a file and displays its content. The filename should be passed as a **command-line argument**.

Solution

```
#include <stdio.h>

#include <stdlib.h>


int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("Usage: %s <filename>\n", argv[0]);
      return 1;
   }


   FILE *file = fopen(argv[1], "r");
   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }


   char ch;
   while ((ch = fgetc(file)) != EOF) {
      putchar(ch);
   }
```

```
    fclose(file);

    return 0;

}
```

Expected Output

```
$ ./readfile example.txt

Hello, this is a test file!
```

Key Learnings

- **Using fopen() and fgetc() to read files**.
- **Passing filenames via command-line arguments**.

CHAPTER 6: ADDITIONAL EXERCISES

Exercise 6: Implement a Queue Using Linked List

Write a **menu-driven** program to implement a **queue** using a linked list.

Exercise 7: Find Prime Numbers

Write a program to **find all prime numbers up to N** using **dynamic memory allocation**.

Exercise 8: Sort an Array Using QuickSort

Implement **QuickSort using recursion** to sort an array.

Conclusion

Hands-on coding practice is **the best way to learn C**. By implementing **basic, intermediate, and advanced exercises,** programmers gain proficiency in:

- **Memory management**
- **Data structures**
- **File handling**
- **Command-line operations**

Assignment Solution: Store and Retrieve Student Records Using Structures in C

Objective

The goal of this assignment is to write a **C program** that stores and retrieves student records using **structures**. The program should allow:

1. **Storing student records** (Name, Roll Number, Marks).
2. **Displaying stored records** from memory.
3. **Using structures efficiently** to organize data.

Step-by-Step Guide

Step 1: Define the Problem

We need to create a **student database** where:

- Each student record consists of **Name, Roll Number, and Marks**.
- The records should be stored using **structures**.
- The program should allow **both storing and retrieving** the records.

Step 2: Plan the Solution

1. **Define a structure (struct Student)** to hold:

- name (String)
- roll_number (Integer)
- marks (Float)

2. **Use an array of structures** to store multiple student records.
3. **Take input for student details**.
4. **Display stored student records**.

STEP 3: IMPLEMENT THE C PROGRAM

Complete Code Implementation

```
#include <stdio.h>

// Step 1: Define a structure for student records
struct Student {
    char name[50];
    int roll_number;
    float marks;
};

int main() {
    int n;
```

```
// Step 2: Ask the user for the number of students
printf("Enter the number of students: ");
scanf("%d", &n);

struct Student students[n]; // Step 3: Create an array of structures

// Step 4: Input student details
for (int i = 0; i < n; i++) {
    printf("\nEnter details for Student %d:\n", i + 1);
    printf("Enter Name: ");
    scanf(" %[^\n]", students[i].name); // Read full name with spaces

    printf("Enter Roll Number: ");
    scanf("%d", &students[i].roll_number);

    printf("Enter Marks: ");
    scanf("%f", &students[i].marks);
}

// Step 5: Display student details
printf("\nStored Student Records:\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("\nStudent %d\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].roll_number);
        printf("Marks: %.2f\n", students[i].marks);
    }

    return 0;
}
```

STEP 4: EXPLANATION OF THE CODE

1. Define the Structure

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

- **name[50]** – Stores the student's full name.
- **roll_number** – Stores the student's unique roll number.
- **marks** – Stores the marks obtained by the student.

---

2. Taking Input for Student Details

```
printf("Enter the number of students: ");

scanf("%d", &n);


struct Student students[n]; // Array of structures
```

- The user enters **how many student records** they want to store.
- An **array of structures** is created dynamically.

```
printf("Enter Name: ");

scanf(" %[^\n]", students[i].name);
```

- **%[^\n]** allows reading full names with spaces.

---

3. Displaying Stored Student Records

```
for (int i = 0; i < n; i++) {
   printf("\nStudent %d\n", i + 1);
   printf("Name: %s\n", students[i].name);
   printf("Roll Number: %d\n", students[i].roll_number);
   printf("Marks: %.2f\n", students[i].marks);
}
```

- The program **loops through all stored records** and prints student details.

STEP 5: EXAMPLE RUNS

Example 1: Input & Output

Input

Enter the number of students: 2

Enter details for Student 1:

Enter Name: Alice Johnson

Enter Roll Number: 101

Enter Marks: 85.5

Enter details for Student 2:

Enter Name: Bob Smith

Enter Roll Number: 102

Enter Marks: 92.0

Output

Stored Student Records:

Student 1

Name: Alice Johnson

Roll Number: 101

Marks: 85.50

Student 2

Name: Bob Smith

Roll Number: 102

Marks: 92.00

STEP 6: ENHANCEMENTS

1. Storing Data in a File

To make student records **persistent**, we can **store and retrieve records from a file**.

Code Modification: Store and Retrieve Records

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

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

void storeRecords(int n, struct Student students[]) {
```

```
    FILE *file = fopen("students.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }

    for (int i = 0; i < n; i++) {
        fprintf(file, "%s %d %.2f\n", students[i].name,
students[i].roll_number, students[i].marks);
    }

    fclose(file);
}

void retrieveRecords() {
    FILE *file = fopen("students.txt", "r");
    if (file == NULL) {
        printf("No records found!\n");
        return;
    }
```

```
    struct Student s;
    printf("\nStored Student Records:\n");
    while (fscanf(file, "%s %d %f", s.name, &s.roll_number, &s.marks)
!= EOF) {
        printf("\nName: %s\nRoll Number: %d\nMarks: %.2f\n", s.name,
s.roll_number, s.marks);
    }

    fclose(file);
}

int main() {
    int choice, n;
    struct Student students[100];

    while (1) {
        printf("\n1. Store Student Records\n2. Retrieve Student
Records\n3. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
        printf("Enter number of students: ");
        scanf("%d", &n);

        for (int i = 0; i < n; i++) {
            printf("\nEnter details for Student %d:\n", i + 1);
            printf("Enter Name: ");
            scanf(" %[^\n]", students[i].name);
            printf("Enter Roll Number: ");
            scanf("%d", &students[i].roll_number);
            printf("Enter Marks: ");
            scanf("%f", &students[i].marks);
        }

        storeRecords(n, students);
        break;
    case 2:
        retrieveRecords();
        break;

    case 3:
```

```
            exit(0);


        default:
            printf("Invalid choice!\n");
    }
  }


  return 0;
}
```

Execution

1. Store Student Records

2. Retrieve Student Records

3. Exit

Enter choice: 1

Enter number of students: 1

Enter details for Student 1:

Enter Name: Alice

Enter Roll Number: 101

Enter Marks: 88.5

1. Store Student Records

2. Retrieve Student Records

3. Exit

Enter choice: 2

Stored Student Records:

Name: Alice

Roll Number: 101

Marks: 88.50

STEP 7: ADDITIONAL EXERCISES

Exercise 1: Add a Search Feature

Modify the program to **search for a student** by roll number.

Exercise 2: Update Student Marks

Allow the user to **update marks** of an existing student.

Exercise 3: Delete a Student Record

Implement a feature to **delete a student record** from the file.

CONCLUSION

By implementing this program, we have:

- **Used structures** to store multiple student records efficiently.

- **Implemented data persistence** using file handling.
- **Created an interactive student management system**.

ASSIGNMENT SOLUTION: DEVELOP A FILE-BASED MINI-DATABASE SYSTEM IN C

Objective

The objective of this assignment is to develop a **file-based mini-database system** that allows users to **add, display, search, and delete records** from a file. This will help in understanding **file handling, structures,** and **memory management** in C.

Step-by-Step Guide to Implementing the Mini-Database System

STEP 1: DEFINE THE STRUCTURE FOR DATA STORAGE

The first step is to define a struct that represents the database record. In this example, we'll create a simple student database that stores:

- **Student ID**
- **Name**
- **Age**
- **Marks**

Structure Definition

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


typedef struct {

   int id;
```

```
    char name[50];
    int age;
    float marks;
} Student;
```

- The typedef keyword makes it easier to use Student instead of struct Student.
- The structure contains fields for **ID, name, age, and marks**.

Step 2: Implement Function to Add a New Record

To add a new student, we write their details to a file using fwrite().

Code to Add a New Record

```
void addStudent() {
    FILE *file = fopen("students.dat", "ab"); // Open in append binary mode
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }

    Student s;
    printf("Enter Student ID: ");
    scanf("%d", &s.id);
    printf("Enter Name: ");
```

```
    scanf(" %[^\n]", s.name); // Read full name with spaces
    printf("Enter Age: ");
    scanf("%d", &s.age);
    printf("Enter Marks: ");
    scanf("%f", &s.marks);

    fwrite(&s, sizeof(Student), 1, file); // Write to file
    fclose(file);

    printf("Student record added successfully!\n");
}
```

- **fopen("students.dat", "ab")**: Opens the file in **append mode** to prevent overwriting.
- **fwrite()**: Writes the structure to the file.

Step 3: Implement Function to Display All Records

To retrieve all records from the database, we read the file using fread().

Code to Display Records

```
void displayStudents() {
    FILE *file = fopen("students.dat", "rb"); // Open in read binary mode
    if (file == NULL) {
        printf("No records found!\n");
```

```
    return;
  }

  Student s;
  printf("\nStudent Records:\n");
  printf("ID\tName\t\tAge\tMarks\n");
  printf("-----------------------------------\n");

  while (fread(&s, sizeof(Student), 1, file)) {
    printf("%d\t%s\t\t%d\t%.2f\n", s.id, s.name, s.age, s.marks);
  }

  fclose(file);
}
```

- **fread()**: Reads records from the file one by one.
- **Loop runs until fread() returns 0**, meaning end-of-file (EOF) is reached.

Step 4: Implement Function to Search for a Record

To find a student by ID, we read each record and compare the ID.

Code to Search for a Student

```
void searchStudent() {
  FILE *file = fopen("students.dat", "rb");
```

```
    if (file == NULL) {
        printf("No records found!\n");
        return;
    }

    int searchId;
    printf("Enter Student ID to search: ");
    scanf("%d", &searchId);

    Student s;
    int found = 0;

    while (fread(&s, sizeof(Student), 1, file)) {
        if (s.id == searchId) {
            printf("\nStudent Found:\n");
            printf("ID: %d\nName: %s\nAge: %d\nMarks: %.2f\n", s.id,
s.name, s.age, s.marks);
            found = 1;
            break;
        }
    }

    if (!found) {
```

```
    printf("Student with ID %d not found!\n", searchId);
  }

  fclose(file);
}
```

- **Scans for a specific ID**.
- **Breaks the loop once found** to improve efficiency.

Step 5: Implement Function to Delete a Record

To delete a record, we:

1. Copy all **records except the one to be deleted** to a new file.
2. Replace the old file with the new one.

Code to Delete a Student

```
void deleteStudent() {
  FILE *file = fopen("students.dat", "rb");
  if (file == NULL) {
    printf("No records found!\n");
    return;
  }

  int deleteId;
  printf("Enter Student ID to delete: ");
```

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

FILE *tempFile = fopen("temp.dat", "wb"); // Temporary file
Student s;
int found = 0;

while (fread(&s, sizeof(Student), 1, file)) {
    if (s.id != deleteId) {
        fwrite(&s, sizeof(Student), 1, tempFile);
    } else {
        found = 1;
    }
}

fclose(file);
fclose(tempFile);

remove("students.dat");     // Delete old file
rename("temp.dat", "students.dat"); // Rename temp file

if (found) {
    printf("Student with ID %d deleted successfully!\n", deleteId);
```

```
    } else {
        printf("Student with ID %d not found!\n", deleteId);
    }
}
```

- Copies all records **except the one to delete**.
- Replaces the old file with the updated one.

STEP 6: IMPLEMENT THE MENU SYSTEM

A menu system provides **user-friendly interaction**.

Main Function

```
int main() {
    int choice;

    do {
        printf("\nMini Database System\n");
        printf("1. Add Student\n");
        printf("2. Display Students\n");
        printf("3. Search Student\n");
        printf("4. Delete Student\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1: addStudent(); break;
            case 2: displayStudents(); break;
            case 3: searchStudent(); break;
            case 4: deleteStudent(); break;
            case 5: printf("Exiting program...\n"); break;
            default: printf("Invalid choice, try again!\n");
        }
    } while (choice != 5);

    return 0;
}
```

- **Uses a do-while loop** to keep showing the menu until the user exits.
- **switch statement** directs the user to the correct function.

Final Output Example

Mini Database System

1. Add Student

2. Display Students

3. Search Student

4. Delete Student

5. Exit

Enter your choice: 1

Enter Student ID: 101

Enter Name: Alice

Enter Age: 20

Enter Marks: 89.5

Student record added successfully!

Enter your choice: 2

Student Records:

ID Name Age Marks

---

101 Alice 20 89.50

Summary of Features

- ✅ **Store Student Records in a File**
- ✅ **Retrieve and Display Data**
- ✅ **Search for a Student**
- ✅ **Delete Student Records**
- ✅ **User-Friendly Menu System**

CONCLUSION

This file-based **mini-database system** demonstrates:

- **File handling (fopen(), fwrite(), fread())**
- **Data storage with structures**
- **Basic database operations (CRUD - Create, Read, Update, Delete)**
- **Memory management and best practices**

Assignment Solution: Implement Memory-Efficient Data Storage Using Dynamic Memory Allocation in C

Objective

The objective of this assignment is to **efficiently store and manage data** in C using **dynamic memory allocation**. This will ensure that memory is allocated as needed, reducing **waste and increasing efficiency**.

Key Concepts Covered:

1. **Understanding malloc(), calloc(), realloc(), and free()**
2. **Efficiently managing memory for variable-sized data**
3. **Avoiding memory leaks by properly freeing allocated memory**
4. **Handling user input dynamically instead of using fixed arrays**

Step-by-Step Guide

Step 1: Define the Problem

A **fixed-size array** wastes memory when storing fewer records than its size and causes **overflow** when exceeding the size. Using **dynamic memory allocation**, we can:

- Allocate memory at **runtime** instead of compile-time.
- Resize memory dynamically if needed.
- Free memory once it's no longer required.

Step 2: Plan the Solution

We will:

1. **Use malloc() to allocate memory for storing student records dynamically.**
2. **Use realloc() to expand memory if additional records need to be stored.**
3. **Use free() to release memory after use.**
4. **Allow the user to input student details dynamically.**

Step 3: Implement the C Program

Complete Code Implementation

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

// Structure to store student details
struct Student {
    char name[50];
    int roll_number;
    float marks;
};

int main() {
    struct Student *students;
```

```
int n, i, new_n;

// Step 1: Get the number of students
printf("Enter the number of students: ");
scanf("%d", &n);

// Step 2: Allocate memory dynamically
students = (struct Student*)malloc(n * sizeof(struct Student));
if (students == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
}

// Step 3: Input student details
for (i = 0; i < n; i++) {
    printf("\nEnter details for Student %d:\n", i + 1);
    printf("Enter Name: ");
    scanf(" %[^\n]", students[i].name);  // Allowing multi-word input
    printf("Enter Roll Number: ");
    scanf("%d", &students[i].roll_number);
    printf("Enter Marks: ");
    scanf("%f", &students[i].marks);
```

```
}

// Step 4: Display stored student details
printf("\nStored Student Records:\n");
for (i = 0; i < n; i++) {
    printf("\nStudent %d\n", i + 1);
    printf("Name: %s\n", students[i].name);
    printf("Roll Number: %d\n", students[i].roll_number);
    printf("Marks: %.2f\n", students[i].marks);
}

// Step 5: Resize memory dynamically (Adding more records)
printf("\nDo you want to add more students? Enter new total count (or same to exit): ");
scanf("%d", &new_n);

if (new_n > n) {
    students = (struct Student*)realloc(students, new_n * sizeof(struct Student));
    if (students == NULL) {
        printf("Memory reallocation failed!\n");
        return 1;
    }
```

```
    // Taking input for new students
    for (i = n; i < new_n; i++) {
        printf("\nEnter details for Student %d:\n", i + 1);
        printf("Enter Name: ");
        scanf(" %[^\n]", students[i].name);
        printf("Enter Roll Number: ");
        scanf("%d", &students[i].roll_number);
        printf("Enter Marks: ");
        scanf("%f", &students[i].marks);
    }
    n = new_n;
}

// Step 6: Display updated records
printf("\nUpdated Student Records:\n");
for (i = 0; i < n; i++) {
    printf("\nStudent %d\n", i + 1);
    printf("Name: %s\n", students[i].name);
    printf("Roll Number: %d\n", students[i].roll_number);
    printf("Marks: %.2f\n", students[i].marks);
}
```

```
    // Step 7: Free allocated memory
    free(students);
    printf("\nMemory freed successfully.\n");

    return 0;
}
```

Step 4: Explanation of the Code

1. Dynamic Memory Allocation (malloc())

```
students = (struct Student*)malloc(n * sizeof(struct Student));
```

- Allocates memory dynamically for **n students**.
- Ensures efficient use of memory **without waste**.
- If allocation fails, NULL is returned, and an error message is displayed.

---

2. Input and Displaying Student Details

```
scanf(" %[^\n]", students[i].name);
```

- scanf(" %[^\n]", students[i].name); ensures **multi-word input** is accepted.
- Loops store multiple student records efficiently.

---

3. Resizing Memory Using realloc()

students = (struct Student*)realloc(students, new_n * sizeof(struct Student));

- Allows expanding storage when more students need to be added.
- Prevents creating **fixed-sized arrays** that waste memory.

---

4. Freeing Allocated Memory (free())

free(students);

- Ensures **no memory leaks** occur.
- Frees memory that is no longer in use.

STEP 5: EXAMPLE RUNS

Example 1: Storing and Displaying Student Records

Input

Enter the number of students: 2

Enter details for Student 1:

Enter Name: Alice Johnson

Enter Roll Number: 101

Enter Marks: 85.5

Enter details for Student 2:

Enter Name: Bob Smith

Enter Roll Number: 102

Enter Marks: 92.0

Output

Stored Student Records:

Student 1

Name: Alice Johnson

Roll Number: 101

Marks: 85.50

Student 2

Name: Bob Smith

Roll Number: 102

Marks: 92.00

Example 2: Expanding Storage and Adding More Records

Input

Do you want to add more students? Enter new total count (or same to exit): 3

Enter details for Student 3:

Enter Name: Charlie Adams

Enter Roll Number: 103

Enter Marks: 78.5

Updated Output

Updated Student Records:

Student 1

Name: Alice Johnson

Roll Number: 101

Marks: 85.50

Student 2

Name: Bob Smith

Roll Number: 102

Marks: 92.00

Student 3

Name: Charlie Adams

Roll Number: 103

Marks: 78.50

Memory freed successfully.

STEP 6: ENHANCEMENTS

1. Sorting Student Records

Modify the program to **sort students by marks** before displaying them.

2. Searching for a Student

Allow users to **search for a student** by roll number.

3. Implementing File Storage

Modify the program to **save records in a file** and **retrieve them later**.

CONCLUSION

This assignment demonstrates **efficient data storage using dynamic memory allocation,** which:

- **Allocates memory dynamically** as per user input.
- **Uses realloc()** to resize storage when needed.
- **Prevents memory wastage** and **avoids fixed-size array limitations**.
- **Properly frees memory (free())** to prevent leaks.