

Assignment 9

Topics: User-defined Datatypes (Structures, Arrays, Pointers), Basics of DMA, and Files

Section A9.1: [Structures]

A9.1a: Define a structure Person with fields name (character array) and age. Write a program to declare and initialize a Person variable, then print the person's details.

A9.1b: Write a program to define a structure Rectangle with fields length and width. Calculate and print the area of the rectangle.

A9.1c: Write a program to define a structure Complex representing complex numbers. Implement the following operations:

1. Subtraction of two complex numbers.
2. Multiplication of two complex numbers.
3. Division of two complex numbers.

A9.1d: Write a program to demonstrate the use of **pointers to structures**. Define a structure Student (Name, Roll Number, Marks), and use a pointer to access the structure's members.

Instructions:

1. Define a Student structure with three fields: name, roll_number, and marks.
2. Declare a pointer to the Student structure.
3. Use the pointer to input and display the details of the student.

Example:

Input: Name: Alice, Roll Number: 102, Marks: 92

Output: Name: Alice, Roll Number: 102, Marks: 92

Hint: Use the arrow operator (->) to access structure members through a pointer.

A9.1e: Write a program to define an **array of structures** for storing book information (Title, Author, Price). Use pointer arithmetic to access and display each book's details.

Instructions:

1. Define a Book structure with three fields: title, author, and price.
2. Declare an array of structures for 3 books.
3. Use pointer arithmetic to traverse and display the details of each book.

Example:

Input:

Book 1: Title: BookA, Author: AuthorA, Price: 250

Book 2: Title: BookB, Author: AuthorB, Price: 300

Output:

Book 1: Title: BookA, Author: AuthorA, Price: 250

Book 2: Title: BookB, Author: AuthorB, Price: 300

Hint: Use pointer arithmetic (book_ptr++) to traverse the array of structures.

A9.1f: Write a program to explore the use of the **sizeof operator on structures**. Define a structure Employee with fields name (character array), age, and salary. Calculate and print the size of this structure and explain why it might be larger than the sum of its parts (due to **structure padding**).

Instructions:

1. Define a structure Employee with fields: name[50], age, and salary.
2. Use the sizeof operator to find the size of the structure and its individual members.
3. Print and compare the sizes.

Example:

Input: -

Output:

Size of Employee structure: 60 bytes

Size of individual members: name = 50, age = 4, salary = 4

Explanation: Padding adds extra bytes to align data in memory.

Hint: The size of a structure may include padding for memory alignment, making the total size larger than the sum of its fields. Also use #pragma pack(n) to customize the padding size and see its effects.

A9.1g: [Bonus] Write a program to demonstrate a **recursive data structure** by defining a Node structure that has an integer data and a pointer to another Node. Create two nodes, link them, and print their values.

Instructions:

1. Define a structure Node with an integer field data and a pointer to Node.
2. Create two nodes dynamically, link them, and display their data.

Example:

Input: Node1 data: 10, Node2 data: 20

Output:

Node1 data: 10

Node2 data: 20

Hint: This program serves as a prelude to linked lists. Use dynamic memory allocation (`malloc()`) to create the nodes.

A9.1h: Write a program to show how **structure variables can be part of another structure**. Define a structure `Address` (`City`, `State`) and a structure `Person` that contains a name and an `Address` structure. Initialize and display the details.

Instructions:

1. Define a structure `Address` with fields: `city` and `state`.
2. Define a structure `Person` with fields: `name` and `Address`.
3. Initialize and print the details of a `Person`, including their address.

Example:

Input: Name: John, City: New York, State: NY

Output: Name: John, Address: New York, NY

Hint: You can nest a structure within another structure using a structure variable.

Section A9.2: [Dynamic Memory Allocation (DMA)]

A9.2a: Write a program to dynamically allocate memory for an array of integers using `malloc()`. Accept values from the user, store them in the allocated memory, and print the elements.

Instructions:

- Use `malloc()` to allocate memory for an array of integers.
- Accept the size of the array from the user.
- Accept the elements of the array from the user.
- Print the array and use `free()` to deallocate the memory.

Example:

Input: 5 (size), [10, 20, 30, 40, 50] (elements)

Output: 10 20 30 40 50

Hint: Use `malloc()` to allocate memory and `free()` to deallocate it after use.

A9.2b: Write a program to dynamically allocate memory for a list of floating-point numbers using `malloc()`. After storing the values, calculate and display their average.

Instructions:

- Use `malloc()` to allocate memory for an array of floats.

- Accept the size and values from the user.
- Calculate and print the average of the elements.
- Deallocate the memory using `free()` after use.

Example:

Input: 4 (size), [1.2, 2.3, 3.4, 4.5] (elements)

Output: Average = 2.85

Hint: Use a loop to calculate the sum, and then divide by the number of elements.

A9.2c [Bonus]: Write a program to dynamically allocate memory for an array of integers using `malloc()`, and then **reallocate** the memory using `realloc()` to store additional integers.

Instructions:

- Allocate memory for an array using `malloc()`.
- Accept the initial size and values from the user.
- Reallocate memory using `realloc()` to expand the array and store additional values.
- Print the final array and use `free()` to deallocate the memory.

Example:

Input: Initial size = 3, [10, 20, 30]

Reallocated size = 5, additional elements [40, 50]

Output: 10 20 30 40 50

Hint: Use `realloc()` to adjust the size of the memory block.

A9.2d: Write a program to demonstrate the use of `calloc()` for dynamically allocating memory for an array of doubles. Input values from the user and print the array.

Instructions:

- Use `calloc()` to allocate memory for an array of doubles.
- Accept the number of elements and input their values.
- Print the elements and deallocate the memory using `free()`.

Example:

Input: 3 (size), [3.14, 2.71, 1.41] (elements)

Output: 3.14 2.71 1.41

Hint: `calloc()` initializes memory to zero by default.

Section A9.3: [Files]

A9.3a: Write a program to **open and close a file** in C. Open the file in write mode, write a simple message to it, and then close the file.

Instructions:

1. Use `fopen()` to open the file in "w" mode (write mode).
2. Use `fprintf()` to write "Hello, World!" to the file.
3. Use `fclose()` to close the file after writing.

Example:

Input: -

Output: File created with message: "Hello, World!"

Hint: Always check if the file was successfully opened by verifying the file pointer is not NULL.

A9.3b: Write a program to **read a file** that contains a single line of text, print its contents, and then close the file.

Instructions:

1. Use `fopen()` to open the file in "r" mode (read mode).
2. Use `fgets()` to read the line from the file and display it.
3. Close the file using `fclose()`.

Example:

Input (file content): "This is a sample text."

Output: "This is a sample text."

Hint: Use `fgets()` to read the line of text into a buffer.

A9.3c: Write a program to explain the **different modes of opening a file** in C. Create a file and open it using different modes: "r", "w", "a", and "r+". Print the behavior of each mode.

Instructions:

1. Create a file and explain how the content is handled when the file is opened in different modes.
2. Open the file in "w" mode, write content, and close it.
3. Then, open the file in "r", "a", and "r+" modes, and explain their behavior.

Example:

Opening in "w": Overwrites content.
Opening in "r": Reads existing content.
Opening in "a": Appends content at the end.
Opening in "r+": Opens the file for both reading and writing.

Hint: The "r+" mode allows both reading and writing but doesn't create a new file if it doesn't exist.

A9.3d: Write a program to **append content to an existing file**. Open the file in append mode and add more text to the end of the file.

Instructions:

1. Open an existing file in "a" mode.
2. Use `fprintf()` to append new text at the end of the file.
3. Close the file after appending the content.

Example:

Input: "New content to append."

Output (file content after append): "This is existing text. New content to append."

Hint: Use the "a" mode to append text without overwriting the existing content.

A9.3e: Write a program to **copy content from one file to another**. Open the source file in "r" mode, and the destination file in "w" mode, then copy all the content.

Instructions:

1. Use `fopen()` to open the source file in "r" mode and the destination file in "w" mode.
2. Read the content from the source file using `fgetc()` or `fgets()` and write it to the destination file.
3. Close both files after copying.

Example:

Input (source file): "Sample text to copy."

Output (destination file): "Sample text to copy."

Hint: Use `fgetc()` to read and write character by character for copying.

A9.3f: Write a program to **write an array of integers to a file** and then read the integers back from the file.

Instructions:

1. Use `fopen()` in "w" mode to write an array of integers to a file using `fprintf()`.

2. Reopen the file in "r" mode to read the integers back and display them.

Example:

Input (array): [1, 2, 3, 4, 5]

Output (from file): 1 2 3 4 5

Hint: Use a loop to write and read the integers one by one.

Section A9.4: [Comprehensive Problems]

A9.4a: Write a program to manage employee records using a structure `Employee` with fields: `name`, `id`, `salary`. Perform the following operations:

1. Add employee records.
2. Display employee records.
3. Save the employee records to a file.
4. Load employee records from the file.

A9.4b: Write a program to maintain a list of products, each having a `name`, `id`, and `price`. Dynamically allocate memory for storing product details and perform the following operations:

1. Add products.
2. Find the product with the highest price.
3. Save the product list to a file.
4. Read the product list from a file.