Name	
Roll No	
Program	BCA
Course Code	DCA1102
Course Name	Programming IN C
Semester	1st

Question .1) Describe various features of the C Programming language .

Answer .:-

Key Features:

- **Structured Programming:**
  - C enforces a structured approach, using control structures like `if`, `else`, `for`, `while`, and `switch` to organize code logically.
  - This enhances readability, maintainability, and reduces errors.
- **Modularity:**
  - C promotes code reusability and organization through functions.
  - Functions encapsulate specific tasks, making code easier to understand, test, and modify.
- **Rich Set of Operators:**
  - C provides a wide range of operators for arithmetic, logical, bitwise, relational, and assignment operations.
  - This allows for flexible and powerful data manipulation.
- **Data Types:**
  - C supports both fundamental (int, float, char, etc.) and derived data types (arrays, pointers, structures, unions).
  - This enables efficient memory usage and representation of various data structures.
- **Memory Management:**
  - C provides manual control over memory allocation and deallocation through functions like `malloc`, `calloc`, `realloc`, and `free`.
  - This allows for efficient memory usage but demands careful management to avoid memory leaks.
- **Pointers:**
  - C offers powerful pointers, variables that store memory addresses.
  - Pointers enable direct memory access, dynamic memory allocation, and efficient data manipulation, but require careful handling to prevent errors.
- **Recursion:**
  - C supports functions calling themselves, creating elegant solutions for problems with self-similar structure (e.g., tree traversals).
  - However, excessive recursion can lead to stack overflow.
- **Portability:**
  - C code can be compiled and run on different platforms with minimal changes, often requiring only recompilation.
  - This makes it a versatile language for cross-platform development.

Other Notable Features:

- **Middle-Level Language:** C bridges the gap between high-level and low-level languages, offering both abstraction and hardware control.
- **Procedural Language:** C focuses on step-by-step execution of instructions, unlike object-oriented languages.
- **Extensibility:** C allows adding new features through libraries and user-defined functions.

Question .2) Explain various flow control statement in C with explains .

Answer .:- Flow control statements govern the order in which instructions are executed, enabling decision-making, repetition, and alteration of the default execution flow.

Types:

1. Selection Statements:

- *if statement:* Executes a code block if a condition is true.

```
Code
if (x > 10) {
   printf("x is greater than 10.\n");
}
```

- **if-else statement:** Chooses between two code blocks based on a condition.

```
Code
if (y == 0) {
   printf("y is zero.\n");
} else {
   printf("y is not zero.\n");
}
```

- **switch statement:** Selects a code block among multiple options based on a variable's value.

```
Code
switch (choice) {
   case 1:
      printf("You chose option 1.\n");
      break;
   case 2:
      printf("You chose option 2.\n");
      break;
   default:
      printf("Invalid choice.\n");
}
```

2. Iteration Statements:

- **for loop:** Executes a code block a specific number of times.

```
Code
for (int i = 0; i < 5; i++) {
   printf("Loop iteration %d\n", i);
}
```

- **while loop:** Executes a code block as long as a condition remains true.

```
Code
```

```
while (num > 0) {
   printf("%d ", num);
   num--;
}
```

- **do-while loop:** Guarantees at least one execution of a code block, then continues based on a condition.

```
Code
do {
   printf("Enter a positive number: ");
   scanf("%d", &num);
} while (num <= 0);
```

3. Jump Statements:

- ***break statement:*** Terminates the execution of a loop or switch statement.
- ***continue statement:*** Skips the remaining code in the current iteration of a loop and jumps to the next iteration.
- ***goto statement:*** Unconditionally jumps to a labeled statement within the same function (generally discouraged for structured programming).

Effective use of flow control statements is essential for writing well-structured, efficient, and adaptable C programs.

Question .3) Define a function List and explain the categories of user-defined functions.
Answer .:-

Function List

- **Definition:** A function list, also known as a function table or function directory, is a central repository that stores information about the available functions within a program. It's often used in compilers and linkers to facilitate function calls and code optimization.

Key Components:

- **Function Name:** The unique identifier used to call the function.
- **Function Address:** The memory location where the function's code resides.
- **Parameter Information:** Data types and order of arguments expected by the function.
- **Return Type:** The data type of the value returned by the function, if any.

Purpose:

- **Function Organization:** Maintains a structured overview of the program's functions.
- **Function Calls:** Enables the compiler to locate and execute the correct function code when a function is invoked.
- **Code Optimization**: Allows the compiler to perform optimizations like function inlining or tail recursion elimination.

- **Runtime Function Resolution:** Used for dynamic linking and loading of libraries during program execution.

Categories of User-Defined Functions

- **Functions Without Arguments and Without Return Values:**
  - Perform tasks without requiring external inputs or producing outputs.
  - Example: void displayWelcomeMessage()
- **Functions Without Arguments but With Return Values:**
  - Calculate and return a value based on internal logic.
  - Example: int generateRandomNumber()
- **Functions With Arguments but Without Return Values:**
  - Receive data through arguments but don't return a value.
  - Example: void printStudentDetails(int rollNo, char name[])
- **Functions With Arguments and With Return Values:**
  - Take inputs and process them to produce a result.
  - Example: float calculateArea(float length, float width)

Additional Considerations:

- **Function Prototypes:** Declarations informing the compiler about function signatures before their definitions.
- **Function Recursion:** Functions calling themselves, creating elegant solutions for certain problems but requiring careful design to avoid stack overflow.
- **Function Overloading:** Multiple functions with the same name but different parameter lists, enhancing code readability and flexibility.

Question .4) Define an array. How to initialize a one – dimensional array ? Explain with suitable example.

Answer .:-

A Foundation for Data Organization

- **Definition:** An array is a fundamental data structure that stores a collection of elements of the same data type, under a single name. It's a structured way to organize and manage multiple values efficiently.
- **Key Characteristics:**
  - **Fixed Size:** Arrays have a predefined size that cannot be changed after declaration.
  - **Sequential Element Storage:** Elements are stored in contiguous memory locations, allowing efficient access using indices.
  - **Same Data Type:** All elements within an array must be of the same data type (e.g., integers, characters, strings).

Initializing One-Dimensional Arrays

- **Declaration and Initialization:**
  - Use this syntax: `data_type array_name[size] = {values};`
    - Example: `int numbers[5] = {10, 20, 30, 40, 50};` creates an integer array `numbers` with 5 elements, initialized with the provided values.
- **Partial Initialization:**
  - **Specify fewer values than the array size:** the remaining elements are initialized to zero.
    - Example: `int scores[10] = {75, 82};` initializes the first two elements, leaving the rest as 0.
- **Initialization Without Values:**
  - **Declare the array with only its size:** elements will have default values (usually zero).
    - Example: `char letters[4];` creates a character array with 4 uninitialized elements.
- **Assigning Values Later:**
  - **Use index notation to assign values to individual elements:**
    `array_name[index] = value;`
    - Example: `numbers[2] = 65;` modifies the third element of `numbers` to 65.

Example in C:

Code

```
#include <stdio.h>

int main() {
   int grades[5] = {85, 92, 78, 64, 90}; // Initializing a 1D array
```

```
    // Accessing and modifying elements:
    grades[3] = 80;  // Changing the fourth grade
    printf("The first grade is: %d\n", grades[0]);
    printf("The average of the grades is: %.2f\n", (double)sum(grades, 5) / 5);

    return 0;
}
```

Essential Points

- Array indices start from 0, not 1.
- Accessing elements beyond the array bounds leads to undefined behavior.
- Choose array sizes carefully to avoid memory issues.
- Arrays are a versatile data structure used extensively in programming for various tasks, including sorting, searching, storing lists, and representing matrices.

Question .5.a) Define Structure and write the general syntax for declaring and accessing members .

Answer .:- Structures: Organizing Related Data Under One Umbrella

Definition: A structure is a user-defined composite data type that allows you to group variables of different data types under a single name. It's like creating a custom blueprint to represent complex data entities cohesively.

Declaring a Structure:

- Use the struct keyword followed by the structure name and a block of member declarations:

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

Example:

Code
```
struct Student {
    int id;
    char name[50];
    float gpa;
};
```

Declaring Structure Variables:

- Create variables of the structure type:

Code
```
struct structure_name variable1, variable2, ...;
```

Accessing Structure Members:

- Use the dot operator (.) to access individual members:

Code

```
variable1.member1 = value;
printf("%s", variable2.name);
```

Example:

Code

```
struct Student student1;
student1.id = 12345;
strcpy(student1.name, "Alice Smith");
student1.gpa = 3.85;
```

Key Points:

- Structures enhance code readability and maintainability by grouping related data.
- Members can have different data types, providing flexibility in data representation.
- Structures are fundamental for building complex data models and organizing code efficiently.

Question .5.b) List out the differences between unions and structures .

Answer .:-

Both unions and structures are user-defined data types in programming that allow you to group related data under a single name. However, they differ in their storage and usage:

Structure:

- Storage: Each member of a structure has its own dedicated memory space. The total size of the structure is the sum of the sizes of its individual members.
- Usage: Suitable for grouping related data that you want to access and manipulate independently.
- Example: A Student structure might have members for name, ID, and GPA. You can access and modify each member separately.

Union:

- Storage: All members of a union share the same memory space. The size of the union is equal to the size of the largest member.
- Usage: Efficient for situations where you only need one member at a time and want to save memory.
- Example: A union might have members for an integer, a character, and a float. Only one of these values can be stored at a time, but it will utilize only the space needed for the largest type.

Here's a table summarizing the key differences:

Feature	Structure	Union
Memory allocation	Separate memory for each member	Shared memory for all members
Size	Sum of member sizes	Size of largest member
Usage	Access and manipulate independent members	Only one member active at a time
Example	Student with name, ID, GPA	Integer, character, float

Choosing between a structure and a union depends on your specific needs. If you need to access and manipulate all members independently, use a structure. If you only need one member active at a time and want to save memory, use a union.

Question .6) Explain the difference between static memory allocation and dynamic memory allocation in C . Explain various dynamic memory allocation function in c .
Answer .:- Static vs. Dynamic
Memory allocation in C determines how memory is assigned to variables and data structures during program execution. It's crucial for efficient memory management and program flexibility.
Static Memory Allocation

- When: Memory is allocated at compile time, before program execution.
- How:
    - Variables declared globally or locally within functions are allocated static memory.
    - Compiler determines memory size based on variable data types.
- **Characteristics:**
    - Fixed size, cannot be changed during runtime.
    - Memory is allocated from the stack segment.
    - Faster as no runtime overhead.
    - Suitable for variables with known sizes in advance.

Dynamic Memory Allocation

- When: Memory is allocated at runtime, during program execution.
- How:
    - Uses standard library functions like malloc(), calloc(), realloc(), and free().
- Characteristics:
    - Allows flexibility in memory allocation as needed.
    - Memory is allocated from the heap segment.
    - Slower due to runtime overhead.
    - Essential for data structures with unknown sizes or requiring resizing.

Dynamic Memory Allocation Functions in C:

1. malloc(size_t size):
    - Allocates a block of memory of specified size (in bytes).
    - Returns a void pointer to the allocated memory, or NULL if allocation fails.
2. calloc(size_t num_elements, size_t element_size):
    - Allocates memory for an array of elements.
    - Initializes all allocated bytes to zero.
3. realloc(void *ptr, size_t new_size):
    - Resizes an existing block of memory previously allocated with malloc() or calloc().
    - Returns a pointer to the resized block, or NULL if resizing fails.
4. free(void *ptr):
    - Deallocates (frees) a previously allocated block of memory, preventing memory leaks.

Choosing the Right Approach:

- Use static allocation for variables with fixed sizes known at compile time.
- Use dynamic allocation for:
    - Data structures with unknown or varying sizes (e.g., linked lists, trees).
    - Allocating memory based on user input or program conditions.
    - Reusing memory blocks to optimize memory usage.

Key Points:

- Careful management of dynamic memory allocation is essential to prevent memory leaks and ensure program stability.
- Always free allocated memory when it's no longer needed.
- Use appropriate dynamic memory allocation functions for different allocation and resizing needs.