black N White

NAME	
ROLL NUMBER	
SEMESTER	1st
COURSE CODE	DCA6110_AUG_2024 MCA_Sem1
COURSE NAME	C PROGRAM LANGUAGE

SET - I

Q1) Describe the basic structure of a C program. Explain scanf() function with an example.

Answer . :- Basic Structure of a C Program
A C program typically consists of the following sections:

1. **Preprocessor Directives:**
    - These directives instruct the preprocessor to modify the source code before compilation.
    - The most common directive is #include, which includes header files that provide definitions for functions and variables.
    - Example:

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

2. **Global Declarations:**
    - Variables declared here are accessible throughout the entire program.
    - However, it's generally recommended to declare variables within functions to improve code organization and avoid potential conflicts.
3. **Main Function:**
    - Every C program must have a main() function, which is the entry point of execution.
    - The main() function returns an integer value to the operating system.
    - Example:

```
int main() {
   // Code goes here
   return 0;
}
```

4. **Function Definitions:**
    - Functions are blocks of code that perform specific tasks.
    - They can be defined before or after the main() function.
    - Example:

```
int add(int a, int b) {
   return a + b;
}
```

The scanf() Function
The scanf() function is a standard input function in C used to read formatted input from the console. It takes a format string and a list of pointers to variables as arguments.
Format String:

- The format string specifies the expected input format, using conversion specifiers to indicate the data type of each input value.
- Common conversion specifiers:
    - %d: Integer
    - %f: Floating-point number
    - %c: Character
    - %s: String

Example:

```
#include <stdio.h>

int main() {
   int age;
   float height;
   char initial;
```

```
    printf("Enter your age, height, and initial: ");
    scanf("%d %f %c", &age, &height, &initial);

    printf("Age: %d\nHeight: %.2f\nInitial: %c\n", age, height, initial);

    return 0;
}
```

Explanation:

1. scanf("%d %f %c", &age, &height, &initial);
   - %d: Reads an integer and stores it in the variable age.
   - %f: Reads a floating-point number and stores it in the variable height.
   - %c: Reads a single character and stores it in the variable initial.
   - The & operator is used to pass the addresses of the variables to scanf(), allowing it to modify the values at those memory locations.
2. printf("Age: %d\nHeight: %.2f\nInitial: %c\n", age, height, initial);
   - This line prints the values of age, height, and initial in a formatted manner.
   - %.2f specifies that the floating-point number height should be printed with two decimal places.

Q.2) What are decision control statements in C? Explain the various types of decision control statements in C programming.

Answer . :- Decision Control Statements in C

Decision control statements in C allow you to alter the flow of execution based on certain conditions. These statements are crucial for creating dynamic and responsive programs. C provides several types of decision control statements:

1. if Statement

The if statement is the most basic decision-making statement. It executes a block of code only if a specified condition is true.

```
if (condition) {
    // Code to be executed if the condition is true
}
```

2. if-else Statement

The if-else statement provides an alternative path of execution if the condition is false.

```
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

3. Nested if-else Statements

You can nest if-else statements to create more complex decision-making structures.

```
if (condition1) {
    // Code to be executed if condition1 is true
} else if (condition2) {
    // Code to be executed if condition1 is false and condition2 is true
} else {
    // Code to be executed if both conditions are false
}
```

4. Switch Statement
The switch statement provides a more efficient way to select one of several code blocks to be executed based on the value of an expression.

```
switch (expression) {
   case value1:
      // Code to be executed if expression == value1
      break;
   case value2:
      // Code to be executed if expression == value2
      break;
   // ...
   default:
      // Code to be executed if no case matches
}
```

How Decision Control Statements Work

1. Condition Evaluation: The program evaluates the condition specified in the decision control statement.
2. Execution Path:
   - If the condition is true, the code within the corresponding block is executed.
   - If the condition is false, the program either moves to the next statement or executes the code in the else block, if present.
3. Flow Control: The decision control statement determines the flow of execution, either continuing with the next statement or jumping to a different part of the code.

Example: Determining Grade Based on Marks

```
#include <stdio.h>

int main() {
   int marks;

   printf("Enter your marks: ");
   scanf("%d", &marks);

   if (marks >= 90) {
      printf("Grade A\n");
   } else if (marks >= 80) {
      printf("Grade B\n");
   } else if (marks >= 70) {
      printf("Grade C\n");
   } else if (marks >= 60) {
      printf("Grade D\n");
   } else {
      printf("Grade F\n");
   }

   return 0;
}
```

In this example, the if-else statements are used to determine the grade based on the input marks. The program evaluates each condition sequentially until a match is found, and then the corresponding grade is printed.

Q.3) What is the purpose of storage classes in C? Discuss the different types of storage classes available in C.

Answer .:- Storage Classes in C
Storage classes in C determine the scope, lifetime, and linkage of variables. They influence how memory is allocated and deallocated for variables, as well as how they can be accessed within a program.
Types of Storage Classes in C

1. **Automatic Storage Class**
   - Scope: Local to the block where it's declared.
   - Lifetime: Exists only while the block is active.
   - Linkage: No linkage.
   - Default: Variables declared inside a function without a storage class specifier are automatically of this type.
   - Example:

```
int main() {
   int x = 10; // x is an automatic variable
   // ...
}
```

2. **External Storage Class**
   - Scope: Global to the entire program.
   - Lifetime: Exists throughout the program's execution.
   - Linkage: External linkage, meaning it can be accessed from other files.
   - Keyword: extern
   - Example:

```
extern int y; // Declare y as an external variable

int main() {
   // ...
}
```

3. **Static Storage Class**
   - Scope: Local to the block where it's declared.
   - Lifetime: Exists throughout the program's execution.
   - Linkage: Internal linkage, meaning it can be accessed within the same file.
   - Keyword: static
   - Example:

```
int main() {
   static int z = 20; // z is a static variable
   // ...
}
```

4. **Register Storage Class**
    - Scope: Local to the block where it's declared.
    - Lifetime: Exists only while the block is active.
    - Linkage: No linkage.
    - Keyword: register
    - Purpose: To suggest to the compiler that the variable should be stored in a register for faster access. However, the compiler may or may not honor this request.
    - Example:

```
int main() {
   register int i; // Suggest i to be stored in a register
   // ...
}
```

SET - II

Q.4) **What is the difference between call by value and call by reference in C? Provide an explanation of recursion with a suitable example.**

Answer . :- Call by Value vs. Call by Reference
In C, function arguments can be passed in two primary ways: call by value and call by reference.

- **Call by Value**
- Mechanism: A copy of the actual argument's value is passed to the function.
- Changes within the Function: Any modifications made to the parameter within the function do not affect the original argument.
- Example:

```
#include <stdio.h>
void swap(int a, int b) {
   int temp = a;
   a = b;
   b = temp;
}
int main() {
   int x = 10, y = 20;
   swap(x, y);
   printf("x = %d, y = %d\n", x, y); // Output: x = 10, y = 20
   return 0;
}
```

In this example, the values of x and y are copied to a and b within the swap function. Swapping a and b does not affect the original values of x and y.

- **Call by Reference**
- Mechanism: The address of the actual argument is passed to the function.
- Changes within the Function: Any modifications made to the parameter within the function directly affect the original argument.
- Implementation in C: C doesn't have direct support for call by reference. However, it can be simulated using pointers.
- Example:

```
#include <stdio.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int x = 10, y = 20;
   swap(&x, &y);
   printf("x = %d, y = %d\n", x, y); // Output: x = 20, y = 10
   return 0;
}
```

In this example, the addresses of x and y are passed to the swap function. The function modifies the values at these addresses, effectively swapping the original values of x and y.

Recursion
Recursion is a programming technique where a function calls itself directly or indirectly. It's often used to solve problems that can be broken down into smaller, self-similar subproblems.

- **Key Components of Recursion:**
  1. **Base Case:** The simplest possible input for which the solution can be directly calculated.
  2. **Recursive Case:** The function calls itself with a smaller input, moving towards the base case.

Example: Factorial Calculation

```
#include <stdio.h>

int factorial(int n) {
   if (n == 0) {
      return 1; // Base case
   } else {
      return n * factorial(n - 1); // Recursive case
   }
}

int main() {
   int num = 5;
   int result = factorial(num);
   printf("Factorial of %d = %d\n", num, result);
   return 0;
}
```

In this example:

1. If n is 0, the function returns 1 (base case).
2. Otherwise, the function recursively calls itself with n-1 and multiplies the result by n.

Recursion is a powerful technique, but it's essential to ensure that the recursive calls eventually reach the base case to avoid infinite recursion.

Q.5) Define pointers in C. Explain pointer arithmetic with an appropriate example.

Answer .:- Pointers in C

A pointer in C is a variable that stores the memory address of another variable. It provides a way to indirectly access and manipulate data. Pointers are powerful tools, but they must be used with caution to avoid potential memory-related issues.

Declaration of a Pointer

To declare a pointer, we use the * operator. For example, to declare a pointer to an integer, we would write:

int *ptr;

This declares a variable ptr that can store the address of an integer.

- **Assigning a Value to a Pointer**

To assign the address of a variable to a pointer, we use the & operator, which gives the address of the variable.

```
int x = 10;

int *ptr = &x;
```

Now, ptr stores the memory address of x.

- **Dereferencing a Pointer**

To access the value stored at the memory address pointed to by a pointer, we use the * operator again, but in a different context. This is called dereferencing the pointer.

```
int y = *ptr;
```

Here, y will be assigned the value of x, which is 10.

Pointer Arithmetic

Pointer arithmetic allows you to perform arithmetic operations on pointers, but it's important to understand that these operations are based on the size of the data type the pointer points to.

For example, if ptr is a pointer to an integer, ptr + 1 will point to the memory location of the next integer, which is 4 bytes away (assuming a 4-byte integer).

```
int arr[5] = {10, 20, 30, 40, 50};

int *ptr = arr;

// Accessing elements using pointer arithmetic

printf("%d %d %d\n", *ptr, *(ptr+1), *(ptr+2)); // Output: 10 20 30
```

Q.6.a) What is the difference between structure and union in C?

Answer . : - Structures and Unions in C

Structures and unions are user-defined data types in C that allow you to group variables of different data types under a single name. However, they have distinct characteristics:

- **Structures**

- Memory Allocation: Each member of a structure occupies its own memory space, regardless of whether it's used or not.
- Access: Members can be accessed individually using the dot (.) operator.
- Use Cases:
  - Representing real-world entities with multiple attributes (e.g., a person with name, age, and address).
  - Organizing related data together for better code readability and maintainability.

Example:

```
struct Person {
   char name[50];
   int age;
   float salary;
};
```

- **Unions**

- Memory Allocation: All members of a union share the same memory space. The size of the union is equal to the size of its largest member.
- Access: Only one member can be active at a time. Accessing a different member overwrites the previous one.
- Use Cases:
  - Saving memory when you need to store different types of data in the same memory location, but only one type is active at a time.
  - Implementing data structures that can represent different states.

- **Example:**

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

- **Key Differences:**

Feature	Structure	Union
Memory Allocation	Individual for each member	Shared by all members
Member Access	Multiple members can be active simultaneously	Only one member can be active at a time
Use Cases	Representing real-world entities, organizing related data	Saving memory, representing different states

In Structures are used to group related data, while unions are used to save memory or represent different data types in the same memory location. The choice between a structure and a union depends on the specific requirements of your application.

Q.6.b) Explain various functions used in dynamic memory allocation.

Answer .:- Dynamic Memory Allocation in C

Dynamic memory allocation in C allows you to allocate memory at runtime, as opposed to static memory allocation, which occurs at compile time. This flexibility is crucial for programs that need to adapt to varying data sizes or complex data structures. C provides several functions for dynamic memory allocation, each with its specific purpose:

1. malloc()

- Purpose: Allocates a block of memory of a specified size in bytes.
- Syntax: void* malloc(size_t size)
- Return Value: A pointer to the allocated memory block, or NULL if allocation fails.

Example:

int *ptr = (int*)malloc(sizeof(int));

2. calloc()

- Purpose: Allocates a block of memory for an array of elements, initializing each element to zero.
- Syntax: void* calloc(size_t num, size_t size)
- Return Value: A pointer to the allocated memory block, or NULL if allocation fails.

Example:

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

3. realloc()

- Purpose: Reallocates a previously allocated memory block to a new size.
- Syntax: void* realloc(void* ptr, size_t new_size)
- Return Value: A pointer to the reallocated memory block, or NULL if allocation fails.

Example:

int *ptr = (int*)malloc(10 * sizeof(int));

// ...

ptr = (int*)realloc(ptr, 20 * sizeof(int));

4. free()

- Purpose: Deallocates a block of memory previously allocated by malloc, calloc, or realloc.
- Syntax: void free(void* ptr)
- Return Value: None

Example:

free(ptr);

<u>Important Considerations:</u>

- Memory Leaks: Always remember to free the memory allocated using these functions to avoid memory leaks.
- Error Handling: Check the return value of these functions to ensure successful allocation. If the return value is NULL, allocation failed.
- Pointer Arithmetic: Be careful when using pointer arithmetic to access elements within the allocated memory block.
- Best Practices: Use these functions judiciously and consider alternative memory management techniques like smart pointers in more complex scenarios.