black N White

NAME	
ROLL NUMBER	
SEMESTER	**1st**
COURSE CODE	**DCA1107_SEP_2024**
COURSE NAME	**BCA**
Subject Name	**C PROGRAMMIG**

SET - I

Q.1) Explain the role of format specifiers in the printf function in C. Provide examples of different format specifiers and their corresponding data types.

Answer : Format Specifiers in C's printf Function

In C programming, the printf function is a versatile tool for formatted output. It allows you to print values to the console in a structured and readable manner. To achieve this, printf employs format specifiers, which are special characters enclosed in percent signs (%) that dictate how the subsequent arguments should be formatted.

Understanding Format Specifiers

A format specifier typically consists of two parts:

1. Percent Sign (%): This signifies the beginning of a format specifier.
2. Conversion Character: This character determines the data type of the argument and the format in which it should be printed.

Common Format Specifiers and Their Data Types

Here are some of the most commonly used format specifiers and their corresponding data types:

Format Specifier	Data Type	Description
%d or %i	int	Signed decimal integer
%u	unsigned int	Unsigned decimal integer
%f	float or double	Floating-point number
%c	char	Single character
%s	char*	String of characters
%p	void*	Pointer address
%x or %X	int or unsigned int	Hexadecimal integer (lowercase or uppercase)
%o	int or unsigned int	Octal integer

Examples of Format Specifiers in Action

```
#include <stdio.h>

int main() {
   int age = 25;
   float pi = 3.14159;
   char initial = 'A';
   char* name = "Alice";

   printf("Age: %d\n", age); // Prints "Age: 25"
   printf("Pi: %.2f\n", pi); // Prints "Pi: 3.14" (2 decimal places)
   printf("Initial: %c\n", initial); // Prints "Initial: A"
   printf("Name: %s\n", name); // Prints "Name: Alice"
   printf("Hexadecimal value of age: %x\n", age); // Prints hexadecimal value of age
   printf("Pointer address of name: %p\n", name); // Prints memory address of name

   return 0;
}
```

Key Points to Remember

- The number of format specifiers in the format string must match the number of arguments passed to the printf function.
- The order of the arguments should correspond to the order of the format specifiers.
- You can customize the output format using flags, field width, and precision specifiers.
- For more complex formatting, consider using the sprintf function to format strings into character arrays.

Q.2) In C programming, what are the decision control statements, and can you give an example of each?

Answer : Decision Control Statements in C

Decision control statements in C programming allow you to alter the flow of execution based on specific conditions. These statements provide the ability to make choices within your program, making it more dynamic and responsive.

Types of Decision Control Statements

1. if Statement

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

```
#include <stdio.h>
int main() {
   int number = 10;
   if (number > 5) {
      printf("Number is greater than 5.\n");
   }
   return 0;
}
```

2. if-else Statement

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

```
#include <stdio.h>
int main() {
   int number = 5;

   if (number > 5) {
      printf("Number is greater than 5.\n");
   } else {
      printf("Number is less than or equal to 5.\n");
   }
   return 0;
}
```

3. Nested if Statements

Nested if statements allow you to create more complex decision-making structures.

```
#include <stdio.h>
int main() {
   int number = 15;
   if (number > 10) {
      if (number < 20) {
         printf("Number is between 10 and 20.\n");
      }
```

```
    }
    return 0;
}
```

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.

```
#include <stdio.h>
int main() {
    int day = 3;

    switch (day) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        default:
            printf("Invalid day\n");
    }
    return 0;
}
```

Key Points to Remember:

- Indentation: Use proper indentation to improve code readability and maintainability.
- Curly Braces: Always use curly braces to enclose the code blocks within if, else, and switch statements, even if they contain only one statement.
- Comparison Operators: Use comparison operators like ==, !=, <, >, <=, and >= to form conditions.
- Logical Operators: Use logical operators like && (AND), || (OR), and ! (NOT) to combine multiple conditions.
- Break Statement: The break statement is used to exit a switch statement or a loop.

Q.3) Explain the concept of arrays in C programming. How are arrays declared and initialized? Discuss with examples.

Answer : Arrays in C Programming

An array in C is a collection of elements of the same data type, stored in contiguous memory locations. It provides a convenient way to store and manipulate multiple values under a single variable name.

Declaration of Arrays

To declare an array, you specify the data type of the elements, the array name, and the number of elements in square brackets:

data_type array_name[array_size];

For example:

int numbers[5]; // Declares an integer array of size 5

float prices[10]; // Declares a float array of size 10

char name[20]; // Declares a character array of size 20

Initialization of Arrays

There are two ways to initialize arrays in C:

1. Static Initialization:

Elements are assigned values at the time of declaration.

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

char name[4] = {'A', 'l', 'i', 'c', 'e'};

2. Dynamic Initialization:

Elements are assigned values after declaration, using a loop or other methods.

```
int i;

int numbers[5];

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

   numbers[i] = i * 10;

}
```

Accessing Array Elements

Array elements are accessed using an index, which starts from 0. For example, to access the third element of the numbers array, you would use numbers[2].

```
printf("The third number is: %d\n", numbers[2]);
```

Key Points to Remember:

- Array indices start from 0, not 1.
- Array size must be a positive integer constant.
- Array elements can be accessed and modified using their indices.
- Array bounds should be checked to avoid out-of-bounds errors.
- Arrays can be passed to functions as arguments.
- Multidimensional arrays can be used to store data in tabular form.

SET - II

Q.4) Explain null-terminated strings in C programming. Discuss how they are different from a regular character array. Provide an example to illustrate your answer.

Answer .: Null-Terminated Strings in C

In C programming, a null-terminated string is a sequence of characters stored in consecutive memory locations, terminated by a null character ('\0'). This null character serves as a sentinel value, indicating the end of the string.

Key Characteristics:

- **Dynamic Length:** Unlike fixed-size arrays, null-terminated strings can have varying lengths, determined by the position of the null character.
- **Memory Allocation:** The memory for a null-terminated string is typically allocated dynamically using functions like malloc or calloc, or statically declared as a character array.
- **String Operations:** C provides a rich set of functions in the string.h library for manipulating null-terminated strings, such as strlen, strcpy, strcat, strcmp, and more.

Difference from Regular Character Arrays:

While both null-terminated strings and regular character arrays store sequences of characters, their key differences lie in their interpretation and usage:

1. **Termination:**
    - **Null-Terminated Strings:** Explicitly terminated by a null character.
    - **Regular Character Arrays:** No implicit termination. The array's size determines the number of characters it can hold.
2. **String Operations:**
    - **Null-Terminated Strings:** Can be manipulated using standard string functions from the string.h library, which rely on the null character to determine string boundaries.
    - **Regular Character Arrays:** Treated as simple arrays of characters. String operations must be implemented manually, often involving loops to iterate over the array elements.
3. **Memory Allocation:**
    - **Null-Terminated Strings:** Often dynamically allocated using malloc or calloc to accommodate varying string lengths.
    - **Regular Character Arrays:** Typically statically declared with a fixed size, limiting their flexibility.

Example:

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

```
#include <string.h>
int main() {
    // Null-terminated string
    char *str1 = "Hello, world!";
    // Regular character array
    char str2[10] = {'H', 'e', 'l', 'l', 'o'};
    // Printing the strings
    printf("Null-terminated string: %s\n", str1);
    printf("Regular character array: ");
    for (int i = 0; i < 5; i++) {
        printf("%c", str2[i]);
    }
    printf("\n");
    // Calculating string lengths
    int len1 = strlen(str1);
    int len2 = sizeof(str2) / sizeof(str2[0]); // Length of the array, not the string
    printf("Length of str1: %d\n", len1);
    printf("Length of str2: %d\n", len2);
    return 0;
}
```

Output:

```
Null-terminated string: Hello, world!
Regular character array: Hello
Length of str1: 13
Length of str2: 10
```

In this example:

- str1 is a null-terminated string. The strlen function can accurately determine its length by locating the null character.
- str2 is a regular character array. It can hold 10 characters, but it doesn't explicitly store a null character. The sizeof operator calculates the total size of the array in bytes, which is then divided by the size of a single character to get the number of elements.

However, this doesn't represent the actual string length, as the array may not be fully utilized.

Null-terminated strings provide a convenient and efficient way to represent and manipulate text in C. By understanding their characteristics and differences from regular character arrays, you can effectively work with strings in your C programs.

Q.5) Explain the concept of recursion in C programming. What are the necessary conditions for a function to be recursive? Provide an example of a recursive function in C that calculates the factorial of a number.

Answer.: Recursion in C Programming

Recursion is a programming technique where a function calls itself directly or indirectly. It's a powerful tool for solving problems that can be broken down into smaller, self-similar subproblems.

Necessary Conditions for Recursion:

1. Base Case: A recursive function must have at least one base case, which is a condition that stops the recursion. Without a base case, the function would call itself infinitely.
2. Recursive Case: The recursive case is where the function calls itself with a simpler version of the problem. This step should bring the problem closer to the base case.

Example: Factorial Calculation

The factorial of a non-negative integer n, denoted as n!, is the product of all positive integers less than or equal to n. It can be defined recursively as follows:

factorial(n) =

1, if n = 0

n * factorial(n-1), otherwise

Here's the C implementation of the recursive factorial function:

```
#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("The factorial of %d is %d\n", num, result);
    return 0;
}
```

How the Recursion Works:

1. Function Call: When factorial(5) is called, the function checks if n is 0. Since it's not, it moves to the recursive case.
2. Recursive Call: It calls factorial(4), which in turn calls factorial(3), and so on, until the base case is reached.
3. Base Case: When factorial(0) is reached, it returns 1.
4. Unwinding the Recursion: The function calls then start returning, multiplying the results at each step:
    - factorial(1) returns 1 * 1 = 1
    - factorial(2) returns 2 * 1 = 2
    - factorial(3) returns 3 * 2 = 6
    - factorial(4) returns 4 * 6 = 24
    - factorial(5) returns 5 * 24 = 120

Key Points to Remember:

- Recursion can make code more elegant and concise, but it can also be less efficient than iterative solutions, especially for large input sizes.
- Excessive recursion can lead to stack overflow errors if the recursion depth exceeds the system's limits.
- It's essential to design recursive functions carefully, ensuring that the base case is reachable and the recursive case moves towards the base case.
- In some cases, tail recursion optimization can be used to improve the efficiency of recursive functions by converting them into iterative form.

While recursion is a powerful tool, it's important to use it judiciously. Consider the trade-offs between recursive and iterative solutions in terms of readability, efficiency, and potential pitfalls.

Q.6) Discuss the selection sort algorithm with an example. Write a C program to implement selection sort.

Answer : Selection Sort Algorithm

Selection sort is a simple sorting algorithm that repeatedly finds the minimum element from the unsorted part of the array and swaps it with the first element of the unsorted part. This process continues until the entire array is sorted.

Algorithm:

1. Find the minimum element: Iterate through the unsorted part of the array to find the minimum element.
2. Swap: Swap the found minimum element with the first element of the unsorted part.
3. Repeat: Repeat steps 1 and 2 for the remaining unsorted part of the array.

Example:

Consider the following unsorted array:

[64, 25, 12, 22, 11]

Pass 1:

- Find the minimum element: 11
- Swap 11 with 64:
- [11, 25, 12, 22, 64]

Pass 2:

- Find the minimum element: 12
- Swap 12 with 25:
- [11, 12, 25, 22, 64]

Pass 3:

- Find the minimum element: 22
- Swap 22 with 25:
- [11, 12, 22, 25, 64]

Pass 4:

- Find the minimum element: 64
- No need to swap, as it's already in the correct position.

The array is now sorted.

C Program Implementation:

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
   int i, j, min_idx;
   // One by one move boundary of unsorted subarray
   for (i = 0; i < n - 1; i++) {
      // Find the minimum element in unsorted array
      min_idx = i;
```

```
        for (j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
// Function to print an array
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted Array\n");
    printArray(arr, n);
    selectionSort(arr, n);
    printf("Sorted array\n");
    printArray(arr, n);
    return 0;
}
```

Time Complexity:

- Best Case: O(n^2)
- Average Case: O(n^2)
- Worst Case: O(n^2)

Space Complexity: O(1)

Selection sort is simple to understand and implement, but it's not very efficient for large datasets due to its quadratic time complexity. It's often used for small datasets or as a teaching tool to introduce sorting algorithms.