**Question: 1 (SET-1)**

**Role of Format Specifiers**

Format specifiers in printf allow the programmer to specify the type and format of the data being printed. They begin with a percent sign (%) and are followed by one or more characters that define the type of the variable being printed. The purpose of using format specifiers is to ensure that each type of data is printed correctly and consistently.

**Common Format Specifiers and Corresponding Data Types**

1. **%d** – **Signed Integer**
   * This specifier is used to print a signed integer (int).
   * Example:

int a = 42;

int b = -42;

printf("The number is: %d\n", a); // Output: The number is: 42

printf("The number is: %d\n", b); // Output: The number is: -42

1. **%u** – **Unsigned Integer**
   * This specifier prints an unsigned integer.
   * That can hold non-negative value.
   * Example:

unsigned int a = 42;

unsigned int b = -42;

printf("The unsigned number is: %u\n", a); // Output: The

unsigned number is: 42

printf("The unsigned number is: %u\n", b); // Output: The

unsigned number is: 4294967254

1. **%f** – **Floating-Point Number**
   * It is used to print a floating-point number (float or double).
   * Example:

float a = 42.5678;

float b = -42.5678;

float c = 43.5678;

printf("The number is: %.2f\n", a); // Output: The number is: 42.57

printf("The number is: %.3f\n", b); // Output: The number is: -42.568

printf("The number is: %f\n", c); // Output: The number is: 43.567799

1. **%c** – **Character**
   * This specifier is used to print a single character.
   * Example:

char a = 'C';

char b = 'Chaitanya';

printf("The letter is: %c\n", a); // Output: The letter is: C

printf("The letter is: %c\n", b); // Output: The letter is: a

1. **%s** – **String**
   * This specifier prints a null-terminated string of characters (a char array).
   * Example:

char a[] = "C";

char b[] = "Chaitanya";

printf("The letter is: %s\n", a); // Output: The letter is: C

printf("The letter is: %s\n", b); // Output: The letter is: Chaitanya

1. **%x** – **Hexadecimal Integer (lowercase)**
   * It prints the integer in hexadecimal (base 16) format, using lowercase letters (a-f).
   * Example:

int num = 255;

printf("Hex value: %x\n", num); // Output: Hex value: ff

1. **%o** – **Octal Integer**
   * This specifier prints the integer in octal (base 8) format.
   * Example:

int num = 64;

printf("Octal value: %o\n", num); // Output: Octal value: 100

1. **%p** – **Pointer**
   * This specifier is used to print the address of a pointer variable.
   * Example:

int num = 10;

int \*ptr = &num;

printf("Address of num: %p\n", ptr); // Output: Address of num: 0x7ffee43b0d0c

**Question: 2 (SET-1)**

The primary decision control statements in C are:

1. **if Statement**
2. **if-else Statement**
3. **else-if Ladder**
4. **switch Statement**

**1. if Statement**

The if statement is the simplest decision-making statement. It executes a block of code if the condition evaluates to true. If the condition is false, the block of code is skipped.

**Syntax:**

if (condition) {

// Code to execute if condition is true

}

**Example:**

int num = 10;

if (num > 5) {

printf("Number is greater than 5\n");

}

In this example, since num is 10 (which is greater than 5), the program will print "Number is greater than 5".

**2. if-else Statement**

The if-else statement is used when there are two possible paths: one to execute if the condition is true, and another if the condition is false.

**Syntax:**

if (condition) {

// Code to execute if condition is true

} else {

// Code to execute if condition is false

}

**Example:**

int num = 3;

if (num > 5) {

printf("Number is greater than 5\n");

} else {

printf("Number is less than or equal to 5\n");

}

Here, since num is 3, the program will print "Number is less than or equal to 5" because the condition num > 5 is false.

**3. else-if Ladder**

The else-if ladder is used when there are multiple conditions to check, and you need to execute a specific block of code depending on which condition is true. It allows for multiple paths, with only one block of code executed.

**Syntax:**

if (condition1) {

// Code for condition1

} else if (condition2) {

// Code for condition2

} else if (condition3) {

// Code for condition3

} else {

// Code if none of the above conditions are true

}

**Example:**

int num = 8;

if (num > 10) {

printf("Number is greater than 10\n");

} else if (num == 8) {

printf("Number is equal to 8\n");

} else {

printf("Number is less than 8\n");

}

In this case, the program will print "Number is equal to 8" because the condition num == 8 is true.

**4. switch Statement**

The switch statement provides a way to execute one out of many possible blocks of code based on the value of a variable or expression. It is often used when there are multiple conditions that depend on a single variable.

**Syntax:**

switch (variable) {

case value1:

// Code to execute if variable == value1

break;

case value2:

// Code to execute if variable == value2

break;

default:

// Code to execute if none of the above cases match

}

**Example:**

int num = 2;

switch (num) {

case 1:

printf("Number is 1\n");

break;

case 2:

printf("Number is 2\n");

break;

case 3:

printf("Number is 3\n");

break;

default:

printf("Number is not 1, 2, or 3\n");

}

Here, since num is 2, the program will print "Number is 2". The break statement is used to exit the switch block once a case is matched.

**Question: 3 (SET-1)**

### Concept of Arrays

An array can store any data type, including integers, floats, and characters. When you declare an array, you need to specify:

1. The type of the elements (e.g., int, float, char).
2. The number of elements the array can hold, which defines its size.

Arrays are particularly useful when you want to perform operations on a large set of data, such as sorting, searching, or performing mathematical calculations on a collection of values.

### Declaring an Array

To declare an array in C, you must specify the type of the array and its size. The syntax is as follows:

type arrayName[size];

* **type**: Specifies the data type of the array elements (e.g., int, char).
* **arrayName**: The name of the array.
* **size**: The number of elements the array can hold.

**Example:**

int numbers[5];

This statement declares an integer array numbers that can hold 5 integer values.

### Initializing an Array

Arrays in C can be initialized either at the time of declaration or later in the program.

#### 1. ****Initialization at the Time of Declaration****

You can initialize an array with specific values when you declare it. If you provide fewer values than the array size, the remaining elements are initialized to zero (for basic data types like int and float).

**Example:**

int numbers[5] = {1, 2, 3, 4, 5};

Here, the array numbers is initialized with 5 integer values. The values are assigned to the array elements as follows:

* numbers[0] = 1
* numbers[1] = 2
* numbers[2] = 3
* numbers[3] = 4
* numbers[4] = 5

#### 2. ****Partial Initialization****

If the number of values provided is less than the size of the array, the remaining elements are set to zero.

**Example:**

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

In this case:

* numbers[0] = 10
* numbers[1] = 20
* numbers[2] = 0 (default value)
* numbers[3] = 0
* numbers[4] = 0

#### 3. ****Implicit Array Size Deduction****

If you don’t specify the size of the array during initialization, C will automatically deduce the size based on the number of elements provided.

**Example:**

int numbers[] = {1, 2, 3, 4, 5};

In this case, the array numbers will have 5 elements, and C will automatically determine the size.

### Accessing Array Elements

Array elements are accessed using their index. Since arrays are zero-indexed, the first element is at index 0, the second at index 1, and so on. You can use square brackets [] to access an element.

**Example:**

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

printf("%d\n", numbers[2]); // Output: 30

**Question: 4 (SET-2)**

**Null-Terminology in Strings**

In C, strings are typically represented as arrays of characters. Each character in the string is stored in consecutive memory locations, and the string ends when a null character ('\0') is encountered. This null character acts as a terminator, signaling the end of the string. The reason this is necessary is that in C, strings are just arrays, and arrays do not inherently store information about their length.

For example, the string "Hello" in C is stored as an array of characters:

arduino

{'H', 'e', 'l', 'l', 'o', '\0'}

Here, the '\0' is essential because it tells functions like printf, strlen, and strcpy where the string ends. Without this null character, there would be no way to determine where the string terminates in memory, leading to undefined behavior or memory access errors.

**Difference Between Null-Terminated Strings and Regular Character Arrays**

A **regular character array** is simply an array of characters without any special indication of where the array ends. It could be used for various purposes, including storing data that isn't necessarily a string (such as a sequence of characters in a file, for example).

A key difference between a null-terminated string and a regular character array is that a **null-terminated string always ends with '\0'** to mark its boundary. This is not the case for a regular character array, which may or may not contain a null character at the end.

Consider the following examples:

* **Null-Terminated String:**

char str[] = "Hello";

* **Regular Character Array:**

char arr[] = {'H', 'e', 'l', 'l', 'o'};

In the first example, str is a null-terminated string, so it has an implicit null character at the end, and functions like strlen(str) would correctly return the length of the string as 5. In the second example, arr is a regular character array, and there is no null character to indicate the end of the array. As a result, functions that expect a string might behave incorrectly when given arr, unless you manually append a null character or specify the length.

**Example Illustrating Null-Terminology vs. Regular Arrays**

Here’s an example illustrating the difference between null-terminated strings and regular character arrays:

#include <stdio.h>

#include <string.h>

int main() {

// Null-terminated string

char str[] = "Hello";

printf("String: %s\n", str); // Output: Hello

printf("Length of string: %zu\n", strlen(str)); // Output: 5

// Regular character array (no null character)

char arr[] = {'H', 'e', 'l', 'l', 'o'};

printf("Character array: %s\n", arr); // Undefined behavior!

printf("Length of character array: %zu\n", strlen(arr)); // Undefined behavior!

return 0;

}

**Explanation:**

1. **Null-Terminated String:**
   * str[] = "Hello"; initializes a null-terminated string with the characters {'H', 'e', 'l', 'l', 'o', '\0'}.
   * The printf function prints the string properly, and strlen correctly calculates the length of the string, returning 5.
2. **Regular Character Array:**
   * arr[] = {'H', 'e', 'l', 'l', 'o'}; initializes a regular character array without the null character at the end.
   * When printf is called with arr, the output is **undefined** because it does not know where the string ends.
   * Similarly, strlen will also behave unpredictably since it tries to find a null character ('\0') to determine the length, but arr doesn't have one.

**Question: 5 (SET-2)**

**Necessary Conditions for a Function to be Recursive**

For a function to be recursive, it must satisfy the following conditions:

1. **Base Case:** Every recursive function must have a base case. The base case is the simplest scenario, which can be solved without further recursive calls. Without a base case, the function would call itself indefinitely, leading to infinite recursion and a stack overflow.
2. **Recursive Case:** The function must call itself with a modified argument that progressively reduces the size of the problem. This recursive call moves the function closer to the base case.
3. **Progressive Reduction:** Each recursive call should bring the function closer to the base case. This often involves reducing the input parameters in a way that simplifies the problem, ensuring that the function will eventually reach the base case.

**Example of a Recursive Function to Calculate Factorial**

A common example of recursion is calculating the **factorial** of a number. The factorial of a number n, denoted n!, is the product of all positive integers from 1 to n. The factorial function is defined as:

* **Base Case:** 0! = 1 (by definition)
* **Recursive Case:** n! = n \* (n-1)! for n > 0

This recursive relation breaks the problem down into progressively smaller subproblems until it reaches the base case.

**Example:**

#include <stdio.h>

// Recursive function to calculate factorial

int factorial(int n) {

// Base case: when n is 0, return 1

if (n == 0) {

return 1;

} else {

// Recursive case: n \* factorial of (n-1)

return n \* factorial(n - 1);

}

}

int main() {

int number = 5;

int result = factorial(number);

printf("Factorial of %d is %d\n", number, result); // Output: Factorial of 5 is 120

return 0;

}

**Explanation:**

1. **Base Case:** The base case is if (n == 0), where the function returns 1. This is the stopping condition for the recursion.
2. **Recursive Case:** If n > 0, the function calls itself with the argument n - 1. This continues until the base case is reached.

For example, when factorial(5) is called, the following sequence of calls occurs:

factorial(5) -> 5 \* factorial(4)

factorial(4) -> 4 \* factorial(3)

factorial(3) -> 3 \* factorial(2)

factorial(2) -> 2 \* factorial(1)

factorial(1) -> 1 \* factorial(0)

factorial(0) -> 1 (base case)

1. The results then "unwind" as the recursive calls return their values:

factorial(1) -> 1 \* 1 = 1

factorial(2) -> 2 \* 1 = 2

factorial(3) -> 3 \* 2 = 6

factorial(4) -> 4 \* 6 = 24

factorial(5) -> 5 \* 24 = 120

The function finally returns 120 as the factorial of 5, which is printed in the main function.

**Question: 6 (SET-2)**

**Selection Sort Algorithm**

**Selection Sort** is a simple comparison-based sorting algorithm that works by repeatedly selecting the minimum (or maximum, depending on sorting order) element from the unsorted part of the array and swapping it with the first unsorted element. This process continues until the entire array is sorted.

The algorithm operates in the following steps:

1. Start from the first element of the array.
2. Search for the smallest element in the unsorted portion of the array.
3. Swap this smallest element with the first unsorted element.
4. Move the boundary between the sorted and unsorted parts of the array by one position (i.e., the first unsorted element is now part of the sorted portion).
5. Repeat the above steps until the unsorted portion becomes empty, indicating the array is sorted.

**Example of Selection Sort**

Consider the following array to be sorted in ascending order:

[64, 25, 12, 22, 11]

1. **First Pass:**
   * The first unsorted part is the entire array: [64, 25, 12, 22, 11].
   * The smallest element is 11. Swap 11 with 64, the first element.
   * After the first pass, the array looks like: [11, 25, 12, 22, 64].
2. **Second Pass:**
   * The unsorted part of the array is [25, 12, 22, 64].
   * The smallest element is 12. Swap 12 with 25.
   * After the second pass, the array looks like: [11, 12, 25, 22, 64].
3. **Third Pass:**
   * The unsorted part is [25, 22, 64].
   * The smallest element is 22. Swap 22 with 25.
   * After the third pass, the array looks like: [11, 12, 22, 25, 64].
4. **Fourth Pass:**
   * The unsorted part is [25, 64].
   * The smallest element is 25. No swap needed.
   * After the fourth pass, the array is: [11, 12, 22, 25, 64].
5. **Fifth Pass:**
   * Only one element, 64, is left, and it is already sorted.

The final sorted array is [11, 12, 22, 25, 64]

**C Program to Implement Selection Sort**

Here is a C program that implements the selection sort algorithm:

#include <stdio.h>

// Function to perform selection sort

void selectionSort(int arr[], int n) {

int i, j, minIdx, temp;

// Traverse through all array elements

for (i = 0; i < n - 1; i++) {

// Find the minimum element in unsorted part of array

minIdx = i;

for (j = i + 1; j < n; j++) {

if (arr[j] < arr[minIdx]) {

minIdx = j;

}

}

// Swap the found minimum element with the first element

if (minIdx != i) {

temp = arr[i];

arr[i] = arr[minIdx];

arr[minIdx] = temp;

}

}

}

// Function to print the array

void printArray(int arr[], int n) {

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

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

}

printf("\n");

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

selectionSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

**Explanation of the Program:**

1. **Function selectionSort:**
   * This function takes an array arr[] and its size n.
   * It uses a loop to traverse the array and find the minimum element in the unsorted portion of the array. It then swaps this minimum element with the first unsorted element.
   * This process repeats for the entire array until the array is sorted.
2. **Function printArray:**
   * This function prints the elements of the array in a readable format.
3. **Main Function:**
   * The main function initializes an array and calls selectionSort to sort it.
   * The array is printed before and after sorting to demonstrate the algorithm.

**Output of the Program:**

Original array: 64 25 12 22 11

Sorted array: 11 12 22 25 64