4. Functions and Scope

1. What is a function in C++? Explain the concept of function declaration, definition, and calling.

- In C++, a function is a block of code that performs a specific task. It's a way to organize code into reusable units, making programs more modular and easier to manage. Functions can take inputs (parameters) and return a value.

- **Function Declaration:**
  A function declaration, also known as a function prototype, informs the compiler about the function's existence before it's used. It specifies the function's name, return type, and the types of its parameters. The declaration doesn't include the actual code that the function executes, only its signature.
  - Syntax:

```
int add(int a, int b); // Function declaration
```

- **Function Definition:**
  A function definition provides the actual implementation of the function. It includes the function's header (same as the declaration) and the function body, which contains the code that the function executes.
  - Syntax:

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

- **Function Calling:**
  A function is called to execute its code. When a function is called, the program control jumps to the function's body, executes the code, and then returns to the point where it was called.
  - Syntax:

```
int result = add(5, 3); // Function call
```

- In summary, the declaration introduces the function, the definition provides the implementation, and calling executes the function.

2. What is the scope of variables in C++? Differentiate between local and global scope.

- In C++, the scope of a variable determines where it can be accessed within the code. There are two primary types of scope: local and global.

- **Local Scope:**
    - Definition:

  Variables declared inside a function, block (defined by curly braces {}), or loop are considered local.

- These variables are only accessible within the specific block where they are defined.
- Their lifetime begins when the block is entered and ends when the block is exited.
- Local variables with the same name can exist in different functions or blocks without causing conflicts.

    - Example:

```
void func() {
   int localY = 100;  // local to func()
   std::cout << localY << "\n";
}

int main() {
   func();
   // std::cout << localY;  // ✕ error: out of scope
}
```

- **Global scope:**
    - Definition**:**

  Variables declared outside of any function or block are considered global.

- They are accessible from any part of the program after their declaration, including inside functions.
- Global variables persist throughout the entire execution of the program.
- Global variables should be used judiciously, as they can make code harder to understand and maintain.

    - Example:

```
int globalX = 42;  // global variable

void printX() {
    std::cout << globalX << "\n";  // accessible here
}

int main() {
    ++globalX;
    std::cout << globalX;         // accessible here too
}
```

❖ Difference:

Feature	Local Scope	Global Scope
Declaration	Inside a function or block.	Outside function or block.
Accessibility	Only within its block.	Throughout the program.
Lifetime	Limited to the block's execution.	Throughout the program execution.
Naming Conflict	Can have same names in different blocks.	Should have unique names.

3. Explain recursion in C++ with an example.

- Recursion in C++ is a programming technique where a function calls itself, either directly or indirectly, to solve a problem. This technique is particularly useful for problems that can be broken down into smaller, self-similar sub-problems.

❖ Key Components of a Recursive Function:

i. **Base Case:**
   This is the condition that stops the recursion. Without a base case, the function would call itself infinitely, leading to a stack overflow.

ii. **Recursive Case:**
   This is where the function calls itself with modified arguments, moving closer to the base case.

➢ **Example**: Calculating Factorial using Recursion

➢ Factorial is a classic recursive example. It's defined mathematically as:
  n! = n × (n-1)!   for n > 0
  0! = 1

➢ The factorial of a non-negative integer n (denoted as n!) is the product of all positive integers less than or equal to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.

```
#include <iostream>

// Recursive function to calculate factorial
int factorial(int n) {
    // Base case: if n is 0 or 1, return 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive case: n * factorial(n-1)
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num;
    std::cout << "Enter a non-negative integer: ";
    std::cin >> num;

    if (num < 0) {
        std::cout << "Factorial is not defined for negative numbers." <<
std::endl;
    } else {
        std::cout << "Factorial of " << num << " is: " << factorial(num)
<< std::endl;
    }

    return 0;
}
```

4. What are function prototypes in C++? Why are they used?

- A function prototype in C++ is a declaration of a function that provides the compiler with information about the function's name, return type, and the types and order of its parameters. It essentially serves as a blueprint for the function, without including the actual function body (the implementation details). A function prototype ends with a semicolon.
- Syntax:

  return_type functionName(parameterType1, parameterType2, ...);

- Example:
  int add(int a, int b); // Function prototype for 'add'

- Function prototypes are used in C++ for several key reasons:

1. **Forward Declaration:**

   They allow you to declare a function before its actual definition appears in the code. This is crucial when functions call each other in a circular manner (e.g., function A calls B, and function B calls A), or when a function is defined later in the same file or in a separate source file.

2. **Type Checking and Error Detection:**

   Function prototypes enable the compiler to perform strong type checking. When a function is called, the compiler can compare the arguments passed to the function with the parameters specified in the prototype. This helps in detecting errors such as:

   - Incorrect number of arguments.
   - Mismatched argument types.
   - Incorrect return type usage.

3. **Code Organization and Readability:**

   Prototypes are often placed in header files (.h or .hpp), which allows for better organization of code. By including the header file, other source files can access and use the declared functions without needing to know their full implementation details, promoting modularity and cleaner code.

4. **Separate Compilation:**

   In larger projects, code is often spread across multiple source files. Function prototypes in header files allow different source files to be compiled independently, as the compiler has the necessary information about the functions even if their definitions reside in other files. This speeds up the compilation process for large projects.