## What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

- Procedural Programming (POP) focuses on functions/procedures.

- OOP focuses on objects which encapsulate data and behavior.

- In POP, data is exposed and can be affected by any function.

- In OOP, data is protected using access modifiers and manipulated through methods.

- POP follows a top-down approach; OOP follows a bottom-up approach.

- POP does not support features like inheritance and polymorphism; OOP supports inheritance, polymorphism, abstraction, and encapsulation.

## List and explain the main advantages of OOP over POP.

- **Modularity**: Code is organized into classes/objects, making it easier to manage.

- **Reusability**: Inheritance allows reusing existing code.

- **Scalability**: Easier to manage and expand.

- **Data Hiding**: Encapsulation protects data from unintended access.

- **Maintenance**: Modular structure simplifies debugging and updates.

- **Real-World Modeling:** Objects represent real-world entities, making programs easier to design and understand.

## 3. Explain the steps involved in setting up a C++ development environment.

- Install a C++ compiler (e.g., GCC or MSVC).

- Install an IDE (e.g., Code::Blocks, Visual Studio, or VS Code).

- Configure the compiler path in the IDE if required.

- Create a new project and write C++ code.

- Compile and run the code using the IDE or terminal.

## 4. What are the main input/output operations in C++? Provide examples.

- Input: Using cin to receive user input.

- Output: Using cout to display output.

Example:

```
#include <iostream>
using namespace std;
int main() {
 int age;
 cout << "Enter your age: ";
 cin >> age;
 cout << "You entered: " << age << endl;
 return 0;
}
```

## What are the different data types available in C++? Explain with examples.

C++ supports several types of data types, which are broadly categorized into the following:

a) Basic Data Types:

- int: Stores integers (e.g., int age = 25;)

- float: Stores floating-point numbers (e.g., float price = 19.99;)

- double: Stores double-precision floating-point numbers (e.g., double pi = 3.14159;)

- char: Stores a single character (e.g., char grade = 'A';)

- bool: Stores boolean values (true or false) (e.g., bool isPassed = true;)

b) Derived Data Types:

- Array (e.g., int numbers[5];)

- Pointer (e.g., int* ptr;)

- Function (e.g., int add(int a, int b);)

c) User-defined Data Types:

- Structure (e.g., struct Student { int id; string name; };)

- Union (e.g., union Data { int i; float f; };)

- Enum (e.g., enum Color { RED, GREEN, BLUE };)

d) Void Data Type:

- Used for functions that do not return a value (e.g., void display();)

## Explain the difference between implicit and explicit type conversion in C++.

- Implicit Type Conversion (Type Promotion):

This is done automatically by the compiler when compatible data types are used.

Example:

```
int x = 10;
double y = x; // int is implicitly converted to double
```

- Explicit Type Conversion (Type Casting):

This is done manually by the programmer using cast operators.

Example:

```
double a = 10.5;
int b = (int)a; // double is explicitly converted to int
```

## What are the different types of operators in C++? Provide examples of each.

a) Arithmetic Operators: +,-, *, /, %

Example: int sum = 5 + 3;

b) Relational Operators: ==, !=, >, <, >=, <=

Example: if (a > b) { ... }

c) Logical Operators: &&, ||, !

Example: if (x > 0 && y > 0) { ... }

d) Assignment Operators: =, +=,-=, *=, /=, %=

Example: x += 5; // same as x = x + 5;

e) Increment/Decrement Operators: ++,--

Example: i++;--j;

f) Bitwise Operators: &, |, ^, ~, <<, >>

Example: int result = a & b;

g) sizeof Operator:

Example: int size = sizeof(int);

## Explain the purpose and use of constants and literals in C++.

- Constants:

Constants are fixed values that do not change during the execution of a program.

Example: const int max_users = 100;

- Literals:

Literals are actual values used in the code.

Types include:

- Integer literal: 100
- Floating-point literal: 3.14
- Character literal: 'A'

- String literal: "Hello"

- Boolean literal: true, false

## What are conditional statements in C++? Explain the if-else and switch statements.

Conditional statements allow you to execute specific blocks of code based on certain conditions.

- if-else Statement:

Used when you want to execute one block of code if a condition is true and another if it is false.

Example:

```
int x = 10;
if (x > 5)
{
cout << "x is greater than 5";
}
else
{
cout << "x is 5 or less";
}
```

- switch Statement:

Used when you have multiple possible values for a variable and want to execute different code based on each value.

Example:

```
int day = 3;
switch(day)
{
case 1: cout << "Monday"; break;
case 2: cout << "Tuesday"; break;
case 3: cout << "Wednesday"; break;
```

```
default: cout << "Invalid day";
}
```

## What is the difference between for, while, and do-while loops in C++?

- for Loop:

Used when the number of iterations is known.

Example:

```
for (int i = 0; i < 5; i++)
{
cout << i << " ";
}
```

- while Loop:

Used when the number of iterations is not known and the condition is checked before executing the loop.

Example:

```
int i = 0;
while (i < 5)
{
cout << i << " ";
i++;
}
```

- do-while Loop:

Similar to while loop but the condition is checked after the loop body, so it runs at least once.

Example:

```
int i = 0;
do {
cout << i << " ";
i++;
```

```
} while (i < 5);
```

## How are break and continue statements used in loops? Provide examples.

- break Statement:

Terminates the loop immediately.

Example:

```
for (int i = 0; i < 10; i++)
{
if (i == 5) break;
cout << i << " ";
}
```

- continue Statement:

Skips the current iteration and continues with the next.

Example:

```
for (int i = 0; i < 5; i++)
{
if (i == 2) continue;
cout << i << " ";
}
```

## Explain nested control structures with an example.

Nested control structures are control statements (like if, loops) inside other control statements.

Example:

```
for (int i = 1; i <= 3; i++)
{
for (int j = 1; j <= 3; j++)
{
```

```
cout << "(" << i << "," << j << ") ";
}
cout << endl;
}
```

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

A function in C++ is a block of code that performs a specific task. Functions help in code reusability and modular programming.

- Function Declaration (Prototype):

It tells the compiler about the function name, return type, and parameters.

Syntax: return_type function_name(parameter_list);

Example: int add(int, int);

- Function Definition:

This contains the actual body of the function.

Example:

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

- Function Calling:

This is how a function is invoked from another function.

Example:

```
int result = add(5, 3);
```

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

Scope refers to the region of the program where a variable is accessible.

- Local Scope:

A variable declared inside a function or block is local to that block.

Example:

```
void func()
{
int x = 10; // x is local to func
}
```

- Global Scope:

A variable declared outside all functions is accessible throughout the program.

Example:

```
int x = 100; // global variable
void show()
{
cout << x;
}
```

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

Recursion is a process where a function calls itself.

Example: Factorial using recursion

```
int factorial(int n)
{
if (n <= 1)
return 1;
else
return n * factorial(n- 1);
}
```

## What are function prototypes in C++? Why are they used?

A function prototype is a declaration of a function that tells the compiler about its return type and parameters.

Example:

int multiply(int, int);

Function prototypes are used to:

- Inform the compiler about the function before its actual definition.

- Enable type checking of arguments during function calls.

- Support top-down programming where function calls can appear before definitions.

## What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

Arrays are collections of elements of the same data type stored in contiguous memory locations.

- Single-dimensional array (1D):

Stores a linear list of elements.

Example:

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

- Multi-dimensional array (e.g., 2D):

Stores elements in a grid (rows and columns).

Example:

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

## Explain string handling in C++ with examples.

C++ handles strings in two main ways:

- C-style strings (character arrays):

Example:

```
char name[10] = "John";
```

- C++ string class (from <string> header):

Example:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
string name = "John";
cout << "Length: " << name.length();
return 0;
}
```

## How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

- 1D Array Initialization:

```
int arr1[5] = {10, 20, 30, 40, 50};
```

- Partial Initialization (rest become 0):

```
int arr2[5] = {1, 2};
```

- 2D Array Initialization:

```
int arr3[2][3] = {
{1, 2, 3},
{4, 5, 6}
};
```

- Partial Initialization of 2D Array:

```
int arr4[2][3] = { {1}, {4, 5} }; // Fills unspecified with 0
```

## Explain string operations and functions in C++

In C++, <string.h> provides functions for handling C-style strings (character arrays). These strings are arrays of characters terminated by a null character '\0'.

1. Declaring a C-style string:

```
char name[20] = "Hello";
```

2. Common Functions from <string.h>:

- strlen(str):

Returns the length of the string (excluding null character).

Example:

```
char name[] = "Coding";
strlen(name); // Output: 6
```

- strcpy(dest, src):

Copies the content of src to dest.

Example:

```
char src[] = "Apple";
char dest[20];
strcpy(dest, src); // dest becomes "Apple"
```

- strcat(dest, src):

Appends src to the end of dest.

Example:

```
char s1[20] = "Hello ";
char s2[] = "World";
strcat(s1, s2); // s1 becomes "Hello World"
```

- strcmp(str1, str2):

Compares two strings.

Returns 0 if equal, <0 if str1 < str2, >0 if str1 > str2.

Example:

```
strcmp("abc", "abc"); // Output: 0
```

- strchr(str, ch):

Returns pointer to first occurrence of character ch in str.

Example:

```
char text[] = "example";
strchr(text, 'm'); // Returns pointer to "mple"
```

Example Program:

```
#include <iostream>
#include <string.h>
using namespace std;
int main()
{
 char name[20] = "Hello";
 char greet[50] = "Welcome ";
 strcat(greet, name);
 cout << "Greeting: " << greet << endl;
 cout << "Length: " << strlen(greet) << endl;
```

```
 return 0;
}
```

## Explain the key concepts of Object-Oriented Programming (OOP).

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects".

The key concepts include:

- Class: Blueprint for creating objects.
- Object: Instance of a class with actual data.
- Encapsulation: Hiding internal details and exposing only necessary parts.
- Abstraction: Hiding complexity and showing only the essential features.
- Inheritance: One class acquiring properties of another class.
- Polymorphism: Ability to use functions or operators in different ways.

## What are classes and objects in C++? Provide an example.

- Class: A user-defined data type that acts as a blueprint for objects.
- Object: An instance of a class.

Example:

```
#include <iostream>
using namespace std;
class Car
{
public:
 string brand;
 int year;
 void display() {
 cout << "Brand: " << brand << ", Year: " << year << endl;
 }
```

```
};
int main() {
 Car myCar;
 myCar.brand = "Toyota";
 myCar.year = 2022;
 myCar.display();
 return 0;
}
```

What is inheritance in C++? Explain with an example.

Inheritance allows a class (derived class) to inherit members from another class (base class).

Example:

```
#include <iostream>
using namespace std;
class Animal
{
public:
 void speak() {
 cout << "This is an animal" << endl;
 }
};
class Dog : public Animal {
public:
 void bark() {
 cout << "Dog barks" << endl;
 }
};
int main() {
 Dog myDog;
```

```
 myDog.speak(); // inherited from Animal
 myDog.bark(); // specific to Dog
 return 0;
}
```

## What is encapsulation in C++? How is it achieved in classes?

Encapsulation is the concept of wrapping data and functions into a single unit (class) and restricting access to the internal details.

It is achieved using access specifiers:

- private: accessible only within the class
- public: accessible from outside the class
- protected: accessible in the class and its derived classes

Example:

```
class Student {
private:
 int marks;
public:
 void setMarks(int m) {
 marks = m;
 }
 int getMarks() {
 return marks;
 }
};
```