Q 1. What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

Feature	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Approach	Top-down	Bottom-up
Focus	Functions & procedures	Objects & classes
Data Handling	Data is exposed to all functions	Data is hidden (encapsulation)
Reusability	Less code reuse	Promotes reusability (inheritance)
Examples	C, Pascal	C++, Java, Python (OOP style)

Q 2. List and explain the main advantages of OOP over POP.

1. **Encapsulation:** Data is bundled with functions, protecting it from unauthorized access.
2. **Reusability:** Through inheritance, existing code can be extended.
3. **Polymorphism:** Same interface can represent different underlying forms (function overloading, overriding).
4. **Maintainability:** Code is organized in modular objects, easier to debug and maintain.
5. **Data Security:** Data hiding ensures that only authorized methods can access data.

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

1. **Install a Compiler:** Install GCC (MinGW on Windows), or Clang on Mac/Linux.
2. **Install an IDE/Text Editor:** Examples - Visual Studio Code, Code::Blocks, Eclipse.
3. **Set Path:** Ensure compiler binaries (like g++.exe) are in your system PATH.
4. **Write Code:** Create .cpp files in your IDE.
5. **Compile:** Use terminal (g++ program.cpp -o program) or IDE build buttons.
6. **Run:** Execute ./program on terminal or use IDE's Run button.

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

- **Input:** cin (console input)
- **Output:** cout (console output)

```
#include <iostream>

using namespace std;

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;
    cout << "You are " << age << " years old.";
    return 0;
}
```

Q 5. What are the different data types available in C++? Explain with examples.

Type	**Example Usage**
int	int x = 5;
float	float y = 5.5;
double	double z = 10.123;
char	char c = 'A';
bool	bool flag = true;
string	string name = "Alice"; (C++ STL)

Q 6. Explain the difference between implicit and explicit type conversion in C++.

- **Implicit Conversion:** Automatic conversion by compiler.

```
int x = 5;
double y = x;  // int to double automatically
```

- **Explicit Conversion:** Done manually by programmer (type casting).

```
double x = 5.5;
int y = (int)x;  // converts double to int
```

Q 7. What are the different types of operators in C++? Provide examples of each.

C++ supports a wide variety of operators. They are classified as follows:

Type	Description & Example
Arithmetic Operators	Used to perform mathematical calculations. Examples: +, -, *, /, % cpp int x = 5 + 3; //8 int y = 10 % 3; //1
Relational Operators	Used to compare two values. Examples: ==, !=, <, >, <=, >= cpp if (a >= b) cout << "a is larger";
Logical Operators	Used for logical operations. Examples: && (AND), `
Assignment Operators	Used to assign values. Examples: =, +=, -=, *=, /=, %= cpp x += 5; // x = x + 5
Increment/Decrement	Increase or decrease value by 1. Examples: ++, -- cpp i++; --j;
Bitwise Operators	Used for bit-level operations. Examples: &, `
Conditional (Ternary)	Shorthand for if-else. Example: condition ? expr1 : expr2 cpp int max = (a > b) ? a : b;
sizeof Operator	Returns size of data type or variable. cpp cout << sizeof(int);
Scope Resolution ::	Used to define a function outside a class or to access global variable. cpp ::x
Member Access . & ->	Used to access members of class or struct. cpp obj.name; ptr->age;

Q 8. Explain the purpose and use of constants and literals in C++.

- **Purpose:** Constants are variables whose values **cannot be changed** after initialization.
- **Why use?**
  - To make programs **more readable**.
  - To **protect data** from accidental modification.
- **How to declare?**

```
const int MAX_USERS = 100;
```

- Trying MAX_USERS = 200; will give a compile-time error.

Literals

- **Purpose:** A literal is a **fixed value** written directly in the code.
- **Types of literals:**
  - **Integer literals:** 10, -25
  - **Floating-point literals:** 3.14, -0.001
  - **Character literals:** 'A', '9'
  - **String literals:** "Hello"
  - **Boolean literals:** true, false

Example combining constants and literals

```
#include <iostream>
using namespace std;
int main() {
  const double PI = 3.14159; // PI is a constant
  double area = PI * 5 * 5;  // 5 is a literal
  cout << "Area: " << area;
  return 0;
}
```

Q 9. What are conditional statements in C++? Explain the if-else and switch statements.

Conditional Statements

- Conditional statements **control the flow of execution** based on conditions (true/false).
- They allow the program to make **decisions**.

🔷 if-else statement

- Used to execute a block of code when a condition is true, and optionally another block when it's false.

```
#include <iostream>
using namespace std;

int main() {
   int age = 18;
   if (age >= 18) {
      cout << "Eligible to vote.";
   } else {
      cout << "Not eligible to vote.";
   }
   return 0;
}
```

🔷 switch statement

- Used when you have multiple specific values to check for a single variable.

```
#include <iostream>
using namespace std;

int main() {
   int day = 3;
   switch(day) {
      case 1: cout << "Monday"; break;
      case 2: cout << "Tuesday"; break;
      case 3: cout << "Wednesday"; break;
      default: cout << "Another day";
   }
   return 0;
}
```

- switch is clearer and faster than many if-else chains when checking **exact matches**.

Q 10. What is the difference between for, while, and do-while loops in C++?

Loop Type	Condition check	Runs at least once?	Typical Use
for	Before loop body	No	Known number of iterations
while	Before loop body	No	Unknown count, runs while condition true
do-while	After loop body	Yes	Run at least once, even if condition false

🔷 **Examples**

```
// for loop
for (int i = 1; i <= 5; i++) {
   cout << i << " "; // 1 2 3 4 5
}

// while loop
int j = 1;
while (j <= 5) {
   cout << j << " ";
   j++;
}

// do-while loop
int k = 1;
do {
   cout << k << " ";
   k++;
} while (k <= 5);
```

Q 11. How are break and continue statements used in loops? Provide examples.

break

- Immediately **exits** the nearest loop.

```
for (int i = 1; i <= 5; i++) {
  if (i == 3) break;
  cout << i << " "; // prints: 1 2
}
```

continue

- **Skips to next iteration** of the loop.

```
for (int i = 1; i <= 5; i++) {
  if (i == 3) continue;
  cout << i << " "; // prints: 1 2 4 5
}
```

Q 12. Explain nested control structures with an example.

Nested control structures

- Control structures **inside another control structure** (like loops inside loops, or if inside loops).

◈ **Example: nested for loops**

```
#include <iostream>
using namespace std;

int main() {
  for (int i = 1; i <= 2; i++) { // outer loop
    for (int j = 1; j <= 3; j++) { // inner loop
      cout << "(" << i << "," << j << ") ";
    }
  }
  return 0;
}
```

Output: (1,1) (1,2) (1,3) (2,1) (2,2) (2,3)

Q 13. What is a function in C++? Explain the concept of function declaration, definition, and calling.

What is a function?

A **function** in C++ is a **block of code that performs a specific task**, can take inputs (parameters), and may return a value. It helps in **modular programming**, improves readability and reusability.

◈ **Parts of a function**

Part	Description	Example
Declaration	Tells the compiler the function's name, return type, and parameters (before main()).	int add(int, int);
Definition	Actual implementation of what the function does.	cpp int add(int a, int b) { return a+b; }
Calling	Executes the function.	sum = add(5, 3);

◈ **Example of all three**

```
#include <iostream>
using namespace std;

// Declaration
int add(int, int);

int main() {
   int result = add(5, 3); // Calling
   cout << "Sum: " << result;
   return 0;
}

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

Q 14. What is the scope of variables in C++? Differentiate between local and global scope.

Scope of variables

- **Scope** defines where a variable can be **accessed or modified**.

❖ Local scope

- Declared **inside a function or block**, accessible only there.

```
void fun() {
   int x = 10; // local variable
   cout << x;
}
// cout << x; // Error: x not accessible here
```

❖ Global scope

- Declared **outside all functions**, accessible anywhere in the program.

```
int y = 20; // global variable

void show() {
   cout << y;
}

int main() {
   cout << y;
   show();
}
```

Scope	Declared	Accessible
Local	Inside function/block	Only inside that function/block
Global	Outside all functions	In all functions

Q 15. Explain recursion in C++ with an example.

What is recursion?

- A function calling **itself directly or indirectly** to solve a problem.
- Useful for problems that can be broken down into similar subproblems (e.g., factorial, Fibonacci).

◈ **Example: factorial using recursion**

```
#include <iostream>
using namespace std;

int factorial(int n) {
  if (n == 0) return 1; // base case
  return n * factorial(n - 1); // recursive call
}

int main() {
  cout << "Factorial of 5 is: " << factorial(5);
  return 0;
}
```

- Here, factorial(5) calls factorial(4), factorial(4) calls factorial(3), and so on until factorial(0).

Q 16. What are function prototypes in C++? Why are they used?

What is a function prototype?

- A function prototype is a **declaration of a function before it is used**.
- It tells the compiler about:
  - The **function name**.
  - **Return type**.
  - **Parameters**.

◈ **Why is it used?**

- Ensures **type checking** before function call.

- Allows you to call a function **before its definition appears** in the code.

🔷 Example

```
#include <iostream>

using namespace std;


// Function prototype

double area(double, double);


int main() {

   cout << "Area: " << area(5.0, 10.0);

   return 0;

}


// Function definition after main

double area(double length, double breadth) {

   return length * breadth;

}
```

Q 17. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

What are arrays?

- An **array** in C++ is a **collection of elements of the same data type**, stored in **contiguous memory locations**.
- It allows you to **store multiple values under a single variable name** using an index.

🔷 Single-dimensional arrays (1D)

- Represented like a list.
- Accessed using **one index**.

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

cout << numbers[2]; // prints 30
```

◈ **Multi-dimensional arrays (2D, 3D,...)**

- Arrays of arrays. Typically used for **matrices or tables**.
- Accessed using **multiple indices**.

```
int matrix[2][3] = {
  {1, 2, 3},
  {4, 5, 6}
};
cout << matrix[1][2]; // prints 6
```

Type	Structure	Access with
1D array	List	arr[i]
2D array	Table (matrix)	arr[i][j]

Q 18. Explain string handling in C++ with examples.

Using C-style strings (char arrays)

- A string is an array of characters ending with a '\0' (null character).

```
char name[6] = "Alice"; // auto adds '\0'
cout << name; // prints Alice
```

- Or initialized manually:

```
char word[] = {'H','i','\0'};
cout << word; // prints Hi
```

Using C++ string class (preferred way)

- Provided by the C++ Standard Library (#include <string>).
- Easier and safer to use.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string greeting = "Hello, World!";
```

```
    cout << greeting << endl;
    greeting += " How are you?";
    cout << greeting;
    return 0;
}
```

Q 19. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Initializing 1D arrays

1. **With explicit values:**

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

2. **Partially initialized (remaining set to 0):**

```
int arr[5] = {1, 2}; // becomes {1,2,0,0,0}
```

3. **Without size (deduced):**

```
int arr[] = {10, 20, 30}; // size = 3
```

Initializing 2D arrays

1. **Direct initialization:**

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

2. **Partial initialization (remaining elements set to 0):**

```
int matrix[2][3] = {1, 2, 3};
// becomes {{1,2,3},{0,0,0}}
```

Q 20. Explain string operations and functions in C++

1. Types of strings in C++

- **C-style strings**: arrays of char, terminated by '\0'.
- **C++ string class**: safer, more powerful, part of the standard library (#include <string>).

Most modern C++ uses the string class.

---

2. Basic string operations using string class

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s1 = "Hello";
    string s2 = "World";

    // Concatenation
    string s3 = s1 + " " + s2;
    cout << s3 << endl; // Hello World

    // Length
    cout << "Length: " << s3.length() << endl;

    // Access characters
    cout << "First char: " << s3[0] << endl;

    // Substring
    cout << "Sub: " << s3.substr(6, 5) << endl; // World

    return 0;
}
```

3. Common string functions

Function	**Description**	**Example**
length() / size()	Number of characters	s.length() → 11
empty()	Checks if string is empty	s.empty()
at(pos)	Returns character at position	s.at(1) → 'e'
substr(pos, len)	Returns substring	s.substr(6,5) → "World"

Function	Description	Example
find(str)	Finds first occurrence of str	s.find("lo") → 3
replace(pos, len, str)	Replaces part with str	s.replace(6,5,"Everyone")
append(str)	Appends string	s.append("!!!") → "Hello!!!"
compare(str)	Compares two strings	s1.compare(s2) returns 0 if equal

Q 21. Explain the key concepts of Object-Oriented Programming (OOP).

OOP is a programming paradigm that organizes software design around **objects** rather than functions and logic.

Key concepts of OOP:

Concept	Description
Class	Blueprint for creating objects (defines attributes & methods).
Object	Instance of a class (real-world entity with state & behavior).
Encapsulation	Bundling data and methods together, restricting direct access.
Abstraction	Hiding complex implementation details, showing only essentials.
Inheritance	Allows a class to acquire properties of another class.
Polymorphism	Same interface, different forms (function overloading & overriding).

Q 22. What are classes and objects in C++? Provide an example.

Class

- A **class** is a **user-defined data type** that acts as a blueprint for objects.
- It contains **data members (variables)** and **member functions (methods)**.

Object

- An **object** is an **instance of a class**.
- Each object has its own copy of data members and can use the class's functions.

◈ **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; // object of class Car
   myCar.brand = "Toyota";
   myCar.year = 2020;
   myCar.display(); // calls member function
   return 0;
}
```

Q 23. What is inheritance in C++? Explain with an example.

Inheritance

- Inheritance allows a **class (derived class / child class)** to **inherit properties and behaviors** (data members & functions) from another **class (base class / parent class)**.
- Promotes **code reuse** and models real-world hierarchies.

◈ **Example**

```
#include <iostream>
using namespace std;

// Base class
class Animal {
```

```
public:
   void eat() {
       cout << "Eating..." << endl;
   }
};

// Derived class
class Dog : public Animal {
public:
   void bark() {
       cout << "Barking..." << endl;
   }
};

int main() {
   Dog d;
   d.eat();  // inherited from Animal
   d.bark(); // own function
   return 0;
}
```

- Dog inherits eat() function from Animal.

Q 24. What is encapsulation in C++? How is it achieved in classes?

Encapsulation

- Encapsulation means **bundling data and functions together in a class**, and **restricting direct access** to some of the object's components.
- It protects data from unintended interference and misuse.

◆ **How is it achieved?**

- Using **access specifiers**: private, protected, public.
    - private: accessible only within the class.

- public: accessible from outside.
- protected: accessible in class and derived classes.

❖ **Example**

```
#include <iostream>
using namespace std;

class Employee {
private:
   int salary; // cannot be accessed directly outside

public:
   void setSalary(int s) {
      salary = s;
   }

   int getSalary() {
      return salary;
   }
};

int main() {
   Employee emp;
   emp.setSalary(50000); // safe access via public function
   cout << "Salary: " << emp.getSalary();
   return 0;
}
```