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

- Programming paradigms define the way a program is structured and developed. Two of the most widely used paradigms are **Procedural Programming** and **Object-Oriented Programming (OOP)**. Both approaches have their own concepts, advantages, and limitations. This assignment explains the key differences between these two programming paradigms.

1. Procedural Programming

- Based on the concept of procedures or functions.
- Follows a top-down approach.
- Program is divided into small parts called functions.
- Data and functions are separate.
- Less secure, as data can be accessed by any function.
- Example languages: C

2. Object-Oriented Programming (OOP)

- Based on the concept of objects and classes.
- Follows a bottom-up approach.
- Program is divided into objects that combine both data and functions.
- Provides features like encapsulation, inheritance, and polymorphism.
- More secure because of data hiding.
- Example languages: C++, Java, Python.

3. Key Differences

Aspect	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Top-down	Bottom-up
Division	Divided into functions	Divided into objects
Focus	Functions	Objects (data + methods)
Data Handling	Data is separate from functions	Data and methods are encapsulated together
Security	Less secure	More secure (encapsulation)
Reusability	Limited (through functions)	High (through inheritance, polymorphism)
Examples	C, Pascal	C++, Java, Python

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

- Programming can be done using different paradigms. Two common paradigms are Procedural-Oriented Programming (POP) and Object-Oriented Programming (OOP). While POP focuses on functions and procedures, OOP focuses on objects that combine both data and functions. OOP provides several advantages over POP, which makes it widely used in modern software development.

Advantages of OOP over POP

1. Modularity (Class and Object Structure)

- In OOP, the program is divided into objects and classes.
- Each class contains its own data and methods, which makes programs more structured and modular.
- In POP, the program is divided into functions only, making it harder to manage for large projects.

2. Data Security (Encapsulation & Data Hiding)

- OOP provides encapsulation, which means data and methods are bundled together.
- Access specifiers (private, protected, public) control how data is accessed.
- POP does not provide proper data security as data is global and can be accessed by any function.

3. Reusability (Inheritance)

- In OOP, existing classes can be reused in new programs through inheritance.
- This avoids rewriting code and improves productivity.
- POP does not support inheritance, so reusability is limited.

4. Flexibility (Polymorphism)

- OOP allows functions or operators to take many forms (polymorphism).
- For example, the same function name can perform different tasks depending on input.
- POP does not support polymorphism, which reduces flexibility.

5. Easy Maintenance and Upgradation

- OOP makes it easy to modify or update code because classes and objects are self-contained.
- Changes in one class do not affect the whole program.
- POP programs are more difficult to maintain because of their top-down structure.

6. Real-World Modeling

- OOP is based on real-world concepts like objects (car, student, bank account).
- This makes it easier to design, understand, and implement large systems.
- POP is more abstract and function-driven, so it does not model real-world entities directly.

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

- Visual Studio Code (VS Code) is a lightweight, powerful, and widely used text editor that supports multiple programming languages, including C++. To use VS Code for C++ development, we need to install a compiler and configure the editor with proper extensions.

Steps to Set Up C++ Development Environment in VS Code

1. Install Visual Studio Code

- Download VS Code from the official website: https://code.visualstudio.com

- Install it on your system (Windows/Linux/Mac).

2. Install a C++ Compiler

- A compiler is required to run C++ programs.
- For Windows: Install MinGW or MSYS2 (provides g++ compiler).
- For Linux: Use terminal command → sudo apt install g++
- For Mac: Install Xcode Command Line Tools → xcode-select --install

3. Add Compiler to System Path (Windows Only)

- After installing MinGW, add its bin folder path (e.g., C:\MinGW\bin) to the Environment Variables → PATH.
- This allows using g++ directly in the terminal.

4. Install C++ Extensions in VS Code

- Open VS Code → Go to Extensions (Ctrl+Shift+X).
- Install the following:
  - C/C++ (by Microsoft) → for IntelliSense, syntax highlighting.
  - C/C++ Compile Run (optional) → to quickly run code.

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

- In C++, Input/Output (I/O) operations are used to take data from the user (input) and display results (output). C++ provides the iostream library which contains the objects cin and cout for input and output operations respectively.

1. Output Operation (cout)

- cout is used to display data on the screen (console output).
- The insertion operator << is used with cout.

2. Input Operation (cin)

- cin is used to take input from the user.
- The extraction operator >> is used with cin.

Example:

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

int main() {
   string name;
   int age;
   cout << "Enter your name and age: ";
   cin >> name >> age;
   cout << "Name: " << name << ", Age: " << age;
```

```
    return 0;
}
```

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

- In C++, **data types** specify the type of data a variable can hold. Data types help the compiler allocate memory and decide how to store and process values. C++ supports several data types, which are classified into different categories.

Types of Data Types in C++

1. Basic Data Types

These are the basic built-in types.

- int → Stores integers (whole numbers).
- float → Stores decimal numbers (single precision).
- double → Stores decimal numbers (double precision).
- char → Stores a single character.
- bool → Stores Boolean values (true or false).
- void → Represents no value (used in functions).

Example:

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

int main() {
    int age = 22;
    float price = 99.50;
    double pi = 3.14159265;
    char grade = 'A';
    bool isStudent = true;

    cout << "Age: " << age << endl;
    cout << "Price: " << price << endl;
    cout << "Value of PI: " << pi << endl;
    cout << "Grade: " << grade << endl;
    cout << "Is Student: " << isStudent;
    return 0;
}
```

2. Derived Data Types

These are built from fundamental data types.

- Arrays → Collection of elements of the same type.
- Pointers → Stores the memory address of another variable.
- Functions → A block of code that performs a task.
- References → Another name for an existing variable.

Example (Array & Pointer):

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

int main() {
   int marks[3] = {85, 90, 78};  // Array
   int *ptr = marks;            // Pointer to first element

   cout << "Marks: " << marks[0] << ", " << marks[1] << ", " << marks[2] << endl;
   cout << "First mark using pointer: " << *ptr;
   return 0;
}
```

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

- In C++, sometimes a variable of one data type is converted into another type. This process is known as type conversion or type casting. There are two types of type conversion in C++:

1. Implicit Type Conversion (Type Promotion / Type Casting by Compiler)
2. Explicit Type Conversion (Type Casting by Programmer)

1. Implicit Type Conversion

- Also called type promotion or type casting by compiler.
- Happens automatically when a smaller data type is assigned to a larger data type.
- No data loss occurs in most cases (but sometimes precision can be lost, e.g., float to int).

2. Explicit Type Conversion

- Also called type casting by programmer.
- The programmer forces a conversion from one type to another.
- Performed using cast operator (type) or functions like static_cast<>.

3. Key Differences

Aspect	Implicit Conversion	Explicit Conversion
Definition	Automatic type conversion done by compiler	Manual type conversion done by programmer
Control	No control by programmer	Full control by programmer
Safety	Generally safe but may lose precision	May cause data loss if used incorrectly
Syntax	No special syntax required	Requires casting operator (type) or static_cast
Example	int a = 10; double b = a;	double pi = 3.14; int x = (int)pi;

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

- Operators in C++ are special symbols used to perform operations on variables and values. C++ provides a wide range of operators that are classified into different categories based on their functionality.

Types of Operators in C++

1. Arithmetic Operators

Used to perform basic mathematical operations.

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus – remainder)

2. Relational (Comparison) Operators

Used to compare two values. Result is either true (1) or false (0).

- == (Equal to)
- != (Not equal to)
- > (Greater than)
- < (Less than)
- >= (Greater than or equal to)
- <= (Less than or equal to)

3. Logical Operators

Used to combine conditions.

- && (Logical AND)
- || (Logical OR)
- ! (Logical NOT)

4. Assignment Operators

Used to assign values to variables.

- = (Assign)
- += (Add and assign)
- -= (Subtract and assign)
- *= (Multiply and assign)
- /= (Divide and assign)
- %= (Modulus and assign)

5. Increment and Decrement Operators

Used to increase or decrease a variable's value by 1.

- ++ (Increment)
- -- (Decrement)

6. Bitwise Operators

Operate on binary representations of integers.

- & (AND)
- | (OR)
- ^ (XOR)
- ~ (NOT / One's complement)
- << (Left shift)
- >> (Right shift)

7. Special Operators

- sizeof → Returns size of a variable/data type.
- typeid → Returns type information (from <typeinfo>).
- comma (,) → Executes multiple expressions.
- scope resolution (::) → Defines scope.

Example:

```
#include <iostream>

using namespace std;


int main() {

   int a = 10, b = 3;


   // 1. Arithmetic Operators

   cout << "Arithmetic (a + b): " << (a + b) << endl;


   // 2. Relational Operators

   cout << "Relational (a < b): " << (a < b) << endl;


   // 3. Logical Operators

   cout << "Logical (a > 0 && b > 0): " << (a > 0 && b > 0) << endl;
```

```
    // 4. Assignment Operators
    int x = 5;
    x += 3;
    cout << "Assignment (x += 3): " << x << endl;

    // 5. Increment/Decrement Operators
    int y = 5;
    cout << "Increment (++y): " << ++y << endl;

    // 6. Bitwise Operators
    cout << "Bitwise (a & b): " << (a & b) << endl;

    // 7. Special Operator
    cout << "Special (sizeof(a)): " << sizeof(a) << endl;

    return 0;
}
```

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

- In C++, data can be stored in two main forms: variables and constants/literals.

- Variables can change their values during program execution.
- Constants and Literals are fixed values that cannot be changed once defined.

They are used to improve readability, reliability, and security in programs.

1. Constants in C++

A constant is a variable whose value cannot be modified after initialization.

- Declared using the const keyword.
- Used to represent fixed values like mathematical constants (PI = 3.14), configuration values, etc.

2. Literals in C++

A literal is a fixed value written directly in the program.
They are also called constants by value.

Types of Literals:

1. Integer Literals → e.g., 10, -25, 0
2. Floating-Point Literals → e.g., 3.14, 2.5e3
3. Character Literals → e.g., 'A', '9'
4. String Literals → e.g., "Hello", "C++"
5. Boolean Literals → true, false
6. Null Literal → nullptr

Difference Between Constants and Literals

Aspect	Constants	Literals
Definition	Named fixed values declared using const.	Direct values written in code.
Changeable?	Cannot be changed once initialized.	Always fixed.
Example	const int MAX = 100;	100, "Hello", 'A'

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

- Conditional statements in C++ are used to make decisions in a program.
  They allow the program to execute different blocks of code based on certain conditions.

Two commonly used conditional statements are:

1. if-else statement
2. switch statement

1. if-else Statement

The if-else statement checks a condition.

- If the condition is true, the if block executes.
- Otherwise, the else block executes.

Syntax:

```
if (condition) {
   // code if condition is true
} else {
   // code if condition is false
}
```

2. switch Statement

The switch statement is used when we need to make a decision among multiple choices.

- It compares a value with different case labels.
- When a match is found, that block executes.
- break is used to exit the switch after execution.
- If no match is found, the default block executes.

Syntex:

```
switch (expression) {
   case value1:
      // code
      break;
   case value2:
      // code
      break;
   default:
      // code if no match
}
```

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

- Loops in C++ are used to repeat a block of code multiple times until a condition is met. The three main types of loops are:

1. for loop
2. while loop
3. do-while loop

Each has a different way of checking the condition and executing the statements.

1. for Loop

- Used when the number of iterations is known in advance.
- Initialization, condition, and increment/decrement are written in one line.

Syntax:

```
for(initialization; condition; update) {
   // code to execute
}
```

2. while Loop

- Used when the number of iterations is not known in advance.
- Condition is checked before executing the loop body.

Syntax:

```
while(condition) {
   // code to execute
}
```

3. do-while Loop

- Similar to while, but the condition is checked after executing the loop body.
- Ensures the loop body runs at least once.

Syntax:

```
do {
   // code to execute
} while(condition);
```

Difference Between for, while, and do-while:

Feature	for Loop	while Loop	do-while Loop
Condition Check	Before execution (pre-test).	Before execution (pre-test).	After execution (post-test).
Usage	Best when number of iterations is known.	Best when number of iterations is unknown.	Best when loop must run at least once.
Syntax	Compact (all in one line).	Condition separate from body.	Condition checked after body.
Minimum Executions	0 times (if condition false).	0 times (if condition false).	1 time (always executes once).

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

- In C++ loops, sometimes we need to control the flow of execution.
  Two special statements used for this purpose are:

1. break statement – used to terminate the loop.
2. continue statement – used to skip the current iteration and move to the next one.

1. break Statement

- The break statement immediately exits the loop when it is encountered.
- The program control moves to the statement after the loop.

Syntax:

```
for (initialization; condition; update) {
   if(condition_to_stop) {
      break;
   }
   // code
}
```

Example (break):

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

int main() {
   for(int i = 1; i <= 10; i++) {
      if(i == 6) {
         break; // exit loop when i = 6
      }
      cout << i << " ";
   }
   return 0;
}
```

2. continue Statement

- The continue statement skips the current iteration and moves to the next iteration of the loop.
- The loop does not terminate; it just ignores the code after continue for that iteration.

Syntax:

```
for (initialization; condition; update) {
   if(condition_to_skip) {
      continue;
   }
   // code
}
```

Example (continue):

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

int main() {
   for(int i = 1; i <= 10; i++) {
      if(i == 6) {
         continue; // skip when i = 6
      }
      cout << i << " ";
   }
   return 0;
}
```

12. Explain nested control structures with an example.

- In C++, control structures (like if, for, while, switch) are used to control the flow of execution. When one control structure is placed inside another, it is called a nested control structure.

Examples include:

- An if statement inside another if.
- A for loop inside another for loop.
- A loop inside an if statement.

Definition

Nested Control Structures are control structures written inside other control structures to handle more complex decision-making or repeated tasks.

Types of Nesting

1. Nested if-else
2. Nested loops (for, while, do-while)

Nested if-else

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

int main() {
   int age = 20;
   char citizen = 'Y'; // Y = Yes, N = No

   if(age >= 18) {
      if(citizen == 'Y') {
         cout << "Eligible to vote";
      } else {
         cout << "Not a citizen, cannot vote";
      }
   } else {
      cout << "Too young to vote";
   }

   return 0;
}
```

Nested for Loop

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

int main() {
   for(int i = 1; i <= 3; i++) {
      for(int j = 1; j <= 3; j++) {
         cout << "(" << i << "," << j << ") ";
      }
      cout << endl;
   }
```

```
    return 0;
}
```

13. 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.
  Instead of writing the same code again and again, we can define a function once and reuse it whenever needed.

Functions help in:

- Code reusability
- Better organization
- Easier debugging and maintenance

Definition

A function is a self-contained block of code that performs a particular task and can be executed when it is called.

Types of Functions in C++

1. Library Functions (predefined, e.g., sqrt(), cout)
2. User-defined Functions (created by the programmer)

Concept of Function in C++

A function in C++ generally has three main steps:

1. Function Declaration (Prototype)

- Tells the compiler about the function name, return type, and parameters.
- Ends with a semicolon (;).

Syntax:

```
return_type function_name(parameter_list);
```

2. Function Definition

- Contains the actual code (logic) of the function.

Syntax:

```
return_type function_name(parameter_list) {
    // body of function
    return value;  // if return type is not void
}
```

3. Function Calling

- A function is executed when it is called inside main() or another function.

Syntax:

function_name(arguments);

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

- In C++, a variable's scope refers to the part of the program where that variable can be accessed or used.
  The scope is important because it controls the lifetime and visibility of a variable in a program.

Types of Variable Scope

1. Local Scope
2. Global Scope

1. Local Scope

- A local variable is declared inside a function or block.
- It is accessible only within that block.
- It is created when the block starts and destroyed when the block ends.

2. Global Scope

- A global variable is declared outside all functions.
- It can be accessed by any function in the program.
- Its lifetime is the entire execution of the program.

Difference between Local and Global Scope:

Feature	Local Variable	Global Variable
Where declared	Inside a function/block	Outside all functions
Visibility	Accessible only within the function/block	Accessible throughout the program
Lifetime	Exists only while the function/block runs	Exists for the entire program execution
Default Value	Garbage (undefined)	Zero (if not initialized)
Priority	Local variable has higher priority if same name as global	Used only if no local variable exists with same name

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

- In C++, recursion is a process in which a function calls itself directly or indirectly to solve a problem.
  It is often used for problems that can be broken into smaller subproblems of the same type (e.g., factorial, Fibonacci series, tree traversal).
- Recursion is a programming technique where a function solves a problem by calling itself on smaller instances of the same problem until it reaches a base case.

Key Components of Recursion

1. Base Case:
   - Condition that stops recursion (prevents infinite calls).
2. Recursive Case:
   - Function calls itself with modified input.

Syntax

```
return_type function_name(parameters) {
   if (base_condition) {
      // base case
      return value;
   } else {
      // recursive case
      return function_name(modified_parameters);
   }
}
```

Example: Factorial using Recursion

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

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

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

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

- In C++, functions can be defined before or after the main() function.
  When a function is defined after main(), the compiler may not recognize it unless we declare its prototype first.
- A function prototype tells the compiler about a function's name, return type, and parameters before its actual definition.
- A function prototype is a declaration of a function that specifies:

- Function name
- Function return type
- Function parameter list

It does not contain the function body.

Syntax

return_type function_name(parameter_list);

Why Function Prototypes are Used?

1. To Inform Compiler
   - The compiler knows about the function before its actual definition.
2. Allows Definition After main()
   - We can place function definitions after main().
3. Helps in Modular Programming
   - Keeps code organized (prototypes in header files, definitions in source files).
4. Error Prevention
   - Ensures correct return type and parameters are used during function calls.

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

- In C++, an array is a collection of elements of the same data type stored in contiguous memory locations.
  Instead of declaring multiple variables for similar data, arrays allow us to store and manage a large amount of data easily.

An array is a data structure that can hold a fixed-size sequence of elements of the same type.
Each element is accessed using an index number (starting from 0).

Syntax

data_type array_name[size];

Types of Arrays

1. Single-Dimensional Array

- Stores data in a single row (linear form).
- Accessed using one index.

2. Multidimensional Array

- Stores data in rows and columns (like a table).
- Accessed using two or more indexes.
- Most common form is the 2D array.

Difference Between Single-Dimensional and Multidimensional Arrays

Aspect	Single-Dimensional Array	Multidimensional Array
Structure	Linear (1 row).	Tabular (rows and columns).
Indexing	Requires one index (e.g., arr[i]).	Requires two or more indexes (e.g., arr[i][j]).
Usage	Best for storing list of values (marks, prices).	Best for storing tabular data (matrices, tables).
Memory Layout	Continuous memory in a single line.	Stored in row-major order (rows stored one after another).

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

- A string is a sequence of characters used to represent text in C++.
  C++ provides two main ways to handle strings:

1. C-Style Strings – Implemented using character arrays and terminated by a null character ('\0').
2. C++ String Class – Part of the Standard Template Library (STL), providing easier and more powerful handling of strings.

1. C-Style Strings

- Declared as an array of characters.
- Always ends with '\0' (null character).
- Operations are performed using functions from the <cstring> library (e.g., strlen, strcpy, strcat).

Example:

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

int main() {
   char str1[20] = "Hello";
   char str2[20] = "World";
```

```
    cout << "Length of str1: " << strlen(str1) << endl;      // string length
    strcat(str1, str2);   // concatenation
    cout << "Concatenated String: " << str1 << endl;

    return 0;
}
```

2. C++ String Class (std::string)

- Declared using the string class from the <string> library.
- Provides built-in operators (+, ==, []) and member functions (length(), substr(), find()).
- Easier and safer than C-style strings.

Example:

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

int main() {
    string name = "Dharmesh";
    string surname = "Lakum";

    string fullname = name + " " + surname;   // concatenation
    cout << "Full Name: " << fullname << endl;

    cout << "Length of Full Name: " << fullname.length() << endl;

    cout << "Substring (0 to 7): " << fullname.substr(0, 7) << endl;

    return 0;
}
```

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

- An **array** in C++ is a collection of elements of the same type stored in contiguous memory.
  Before using an array, it must be **declared** and can be **initialized** (assigned values).

C++ allows different ways of initializing arrays at the time of declaration or later in the program.

1. Initialization of One-Dimensional (1D) Arrays

A 1D array stores elements in a single row (linear form).

Syntax:

```
data_type array_name[size] = {value1, value2, ..., valueN};
```

Examples:

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

int main() {
    // Method 1: Full initialization
    int arr1[5] = {10, 20, 30, 40, 50};

    // Method 2: Partial initialization (remaining values become 0)
    int arr2[5] = {1, 2};   // arr2 = {1, 2, 0, 0, 0}

    // Method 3: Compiler automatically determines size
    int arr3[] = {5, 10, 15, 20};

    cout << "Array 1: ";
    for (int i = 0; i < 5; i++) cout << arr1[i] << " ";

    cout << "\nArray 2: ";
    for (int i = 0; i < 5; i++) cout << arr2[i] << " ";

    cout << "\nArray 3: ";
    for (int i = 0; i < 4; i++) cout << arr3[i] << " ";

    return 0;
}
```

2. Initialization of Two-Dimensional (2D) Arrays

A 2D array is like a table (rows and columns).

Syntax:

```
data_type array_name[rows][columns] = {
   {value1, value2, ...},
   {value3, value4, ...}
};
```

Examples:

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

int main() {
    // Method 1: Full initialization
    int matrix1[2][3] = { {1, 2, 3}, {4, 5, 6} };

    // Method 2: Row-wise initialization
    int matrix2[2][3] = {1, 2, 3, 4, 5, 6};

    // Method 3: Partial initialization (missing values become 0)
    int matrix3[2][3] = { {7, 8}, {9} };   // = {{7, 8, 0}, {9, 0, 0}}
```

```
    cout << "Matrix 1:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix1[i][j] << " ";
        }
        cout << endl;
    }

    cout << "\nMatrix 3:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix3[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

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

- C++ provides many operations and functions for handling strings efficiently.

a) Concatenation (Joining two strings)

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

int main() {
    string s1 = "Hello ";
    string s2 = "World!";
    string s3 = s1 + s2;   // Concatenation
    cout << s3;
    return 0;
}
```

b) Length of a String

```
string s = "Programming";
cout << "Length = " << s.length();
```

c) Accessing Characters

```
string s = "C++";
cout << s[0];   // Prints 'C'
```

d) Comparing Strings

```
string a = "Apple";
string b = "Banana";
if(a == b)
    cout << "Equal";
```

```
else
   cout << "Not Equal";
```

e) Substring Extraction

```
string s = "HelloWorld";
cout << s.substr(0, 5);  // From index 0, take 5 characters
```

f) Inserting into String

```
string s = "C++ is fun";
s.insert(4, "programming ");
cout << s;
```

g) Erasing Part of a String

```
string s = "OpenAI ChatGPT";
s.erase(6, 4);   // Remove 4 characters from index 6
cout << s;
```

h) Finding Substring

```
string s = "I love coding in C++";
int pos = s.find("coding");
cout << "Found at position: " << pos;
```

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

- Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, instead of just functions and logic.

- An object represents a real-world entity (like a student, car, or bank account).
- OOP is based on the concept of classes and objects, making programs more modular, reusable, and easier to maintain.

Key Concepts of OOP

1. Class

- A class is a blueprint for creating objects.
- It defines attributes (data members) and behaviors (member functions).

2. Object

- An object is an instance of a class.
- It represents a specific entity with unique values.

3. Encapsulation

- Wrapping data and methods together inside a class.

- Provides data hiding using access specifiers (private, public, protected).

4. Abstraction

- Hiding implementation details and showing only the essential features.
- Achieved using abstract classes and interfaces.

5. Inheritance

- Allows a class (child/derived) to acquire properties and behaviors of another class (parent/base).
- Promotes reusability.

6. Polymorphism

- Ability of a function or operator to behave differently based on context.
- Two types:
    - Compile-time (Function/Operator Overloading)
    - Run-time (Function Overriding with virtual functions)

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

- In C++, Object-Oriented Programming (OOP) is based on the concept of classes and objects.

- A class is a blueprint or template that defines the structure and behavior of objects.
- An object is an instance of a class that represents a real-world entity.

1. Class in C++

- A class groups data members (variables) and member functions (methods) together.
- Declared using the keyword class.
- Access specifiers like public, private, and protected are used to control accessibility.

Syntax:

```
class ClassName {
   // data members
   // member functions
};
```

2. Object in C++

- An object is created from a class.
- It can access public members of the class using the dot (.) operator.

Syntax:

```
ClassName objectName;
```

Example

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

// Defining a Class
class Student {
public:
   string name;
   int age;

   // Member function
   void display() {
      cout << "Name: " << name << endl;
      cout << "Age: " << age << endl;
   }
};

int main() {
   // Creating an Object
   Student s1;
   s1.name = "Dharmesh";
   s1.age = 20;

   // Calling member function
   s1.display();

   return 0;
}
```

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

- Inheritance is one of the key features of Object-Oriented Programming (OOP) in C++.

- It allows a class (child/derived class) to acquire the properties and behaviors of another class (parent/base class).
- The main purpose of inheritance is code reusability and establishing relationships between classes.

Definition of Inheritance

- Inheritance is a mechanism in C++ by which one class can derive or extend the properties (data members) and behaviors (member functions) of another class.
- The class being inherited from is called the Base Class.
- The class that inherits is called the Derived Class.

Types of Inheritance in C++

1. Single Inheritance – One base class and one derived class.

2. Multiple Inheritance – One class inherits from more than one base class.
3. Multilevel Inheritance – A class is derived from another derived class.
4. Hierarchical Inheritance – Multiple classes are derived from a single base class.
5. Hybrid Inheritance – A combination of two or more types.

Example

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

// Base Class
class Vehicle {
public:
   void move() {
      cout << "This vehicle can move." << endl;
   }
};

// Derived Class
class Car : public Vehicle {
public:
   void brand() {
      cout << "Brand: Toyota" << endl;
   }
};

int main() {
   Car c1;

   // Accessing base class function
   c1.move();

   // Accessing derived class function
   c1.brand();

   return 0;
}
```

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

- Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP) in C++.
  It is the process of wrapping data (variables) and methods (functions) into a single unit called a class.
  It helps in data hiding and protecting data from unauthorized access.

Definition of Encapsulation

Encapsulation is the mechanism of bundling data members and member functions together inside a class, while restricting direct access to some of the object's components.

How Encapsulation is Achieved in C++?

1. Using Classes – Data and methods are combined inside a class.
2. Access Specifiers – private, public, and protected are used to control access:
    - private → data is hidden, accessible only inside the class.
    - public → data/methods accessible outside the class.
    - protected → accessible in derived classes.
3. Getters and Setters – Public functions are used to read or modify private data safely.