Software Engineering Assignment

MODULE: 3

Introduction to OOPS Programming

1. Introduction to C++

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

=> The key differences between **Procedural Programming** and **Object-Oriented Programming (OOP)** lie in their fundamental paradigms, structure, and approach to problem-solving. Here's a breakdown:

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called ***functions***.	In object-oriented programming, the program is divided into small parts called ***objects***.
Procedural programming follows a ***top-down approach.***	Object-oriented programming follows a ***bottom-up approach***.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented programming provides data hiding so it is ***more secure***.

Procedural programming is based on the ***unreal world***.	Object-oriented programming is based on the ***real world***.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.

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

=>

1.Enhanced Modularity and Code Reusability

OOP divides a program into smaller, independent, and self-contained units called **classes**.

Each class encapsulates its own data and methods, making it easier to work on individual components without affecting others.

Reusability is achieved through features like **inheritance** (reuse functionality in derived classes) and **composition** (combine objects to create complex systems).

Example:

Once a class is created, it can be reused in multiple programs or projects with minimal modifications.

2.Encapsulation for Data Protection

Encapsulation binds data (attributes) and behavior (methods) together within an object. It hides the internal state of an object and exposes only the necessary details via public methods.

This prevents direct access to sensitive data, reducing the risk of accidental corruption or unauthorized access.

Example: By using private or protected access modifiers, you can restrict access to certain parts of a class.

3.Better Scalability for Larger Systems

OOP's hierarchical structure, modular design, and support for abstraction make it easier to manage and scale large applications. New functionality can be added without significantly altering existing code.

Example:

New features can be implemented by adding or modifying specific classes without needing to rewrite the entire program.

4.Easier Maintenance and Debugging

OOP simplifies maintenance by isolating changes to specific classes or objects. Debugging is also easier because each object's behavior is independent, making it simpler to trace issues.

Example:

If a bug is found in a class, fixing it does not affect unrelated parts of the program.

5.Closer Representation of Real-World Problems

OOP uses objects to represent real-world entities and their interactions.

This makes programs more intuitive and aligns the design with real-world scenarios, improving readability and comprehension.

Example:

A `Car` object with attributes like `speed` and `color` and methods like `start()` and `stop()` closely mirrors a real-world car.

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

=> Setting up a C++ development environment involves a series of steps to ensure you have the tools required to write, compile, and run C++ programs efficiently. Here's a step-by-step guide:

1.Install a C++ Compiler

A compiler is essential for converting your C++ code into executable programs. Common compilers include:

GCC (GNU Compiler Collection): Open-source and widely used.
MSVC (Microsoft Visual C++): Part of the Visual Studio IDE for Windows.
Clang: A modern, fast, and flexible compiler.
Steps to Install GCC or Clang:
On **Windows**:
Install **MinGW** or **MSYS2** for GCC or Clang.

Alternatively, use **Cygwin** or **WSL (Windows Subsystem for Linux)**.
On **macOS**:
Install Xcode Command Line Tools using:

2. Choose and Install an Integrated Development Environment (IDE) or Text Editor

An IDE or text editor streamlines coding with features like syntax highlighting, debugging, and code suggestions.

Popular IDEs for C++:

Visual Studio (Windows): Comprehensive and beginner-friendly.
CLion: A cross-platform IDE from JetBrains with powerful features.
Code::Blocks: Lightweight and cross-platform.
Eclipse CDT: Open-source and extensible.

Popular Text Editors:
VS Code (Visual Studio Code):
Requires the **C/C++ Extension** from the marketplace.
Sublime Text:
Add plugins like **SublimeClang** for C++ support.
Vim/Neovim:
Configure with plugins like YouCompleteMe or coc.nvim for autocompletion.

3. Set Up the Compiler in Your IDE or Text Editor

After installing an IDE or text editor, you need to configure it to use the C++ compiler:

Visual Studio: Built-in support for MSVC. During installation, select the "Desktop Development with C++" workload.

VS Code:
Install the **C/C++ Extension**.
Configure the `tasks.json` and `launch.json` files to set up build and debug tasks.
Code::Blocks:
Detects compilers like GCC during installation; otherwise, configure paths in **Settings > Compiler**.

4. Install a Debugger

A debugger helps you identify and fix errors in your code. Common choices:

GDB (GNU Debugger): Works with GCC and Clang.
LLDB: Default debugger for Clang.
Microsoft Debugger: Included with Visual Studio.

Most IDEs integrate debugging support directly with the debugger, so minimal configuration is needed.

5. Write Your First C++ Program

Once the tools are installed, write a simple program to test the setup

Example Code

```
#include <iostream>

using namespace std;

 main() {

   cout << "Hello, World!" << endl;

   return 0;

}
```

6.Compile and Run Your Program

Open a terminal or command prompt, navigate to the file directory, and compile the program
g++ hello.cpp -o hello
Run the executable:
./hello # On macOS/Linux
hello.exe # On Windows

7.Verify and Test the Environment
Ensure the compiler version is correct by running:

```
Test debugging by setting breakpoints in your IDE or using GDB

gdb ./hello
```

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

=> In C++, input and output (I/O) operations are performed using the **standard input/output streams** provided by the Standard Library. These streams are part of the `<iostream>` header. The main I/O operations in C++ include.

1. Standard Input (cin)

Used to read data from the user via the keyboard.

Operates through the `std::cin` object.

Syntax:

```
std::cin >> variable;
```

Example: Reading Input

```
#include <iostream>
using namespace std;
 main()
{
   int age;
   cout << "Enter your age: ";
   cin >> age;  // Takes input and stores it in 'age'
   cout << "You entered: " << age << endl;

}
```

2. Standard Output (cout)

Used to display data on the screen.

Operates through the `std::cout` object.

Syntax:

```
std::cout << data;
```

Example: Displaying Output

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

```
main() {
      cout << "Hello, World!" << endl;  // Outputs a message followed
by a newline
}
```

3. Standard Error (cerr)

Used to output error messages.
Operates through the `std::cerr` object.
Does not buffer the output (i.e., it's immediate).

Example: Displaying Errors

```
#include <iostream>
using namespace std;
main()
{
   cerr << "An error occurred!" << endl;  // Outputs an error message

}
```

4. Standard Log (clog)

Used for logging messages.
Operates through the `std::clog` object.
Buffered (unlike `cerr`).

Example: Logging Messages

```
#include <iostream>
using namespace std;
main()
```

```
{
   clog << "This is a log message." << endl;  // Outputs a log message
   return 0;
}
```

5. File Input/Output

For reading and writing files, C++ uses the `<fstream>` library, which provides:

ifstream: Input file stream for reading from files.
ofstream: Output file stream for writing to files.
fstream: For both reading and writing.

Example file i/o

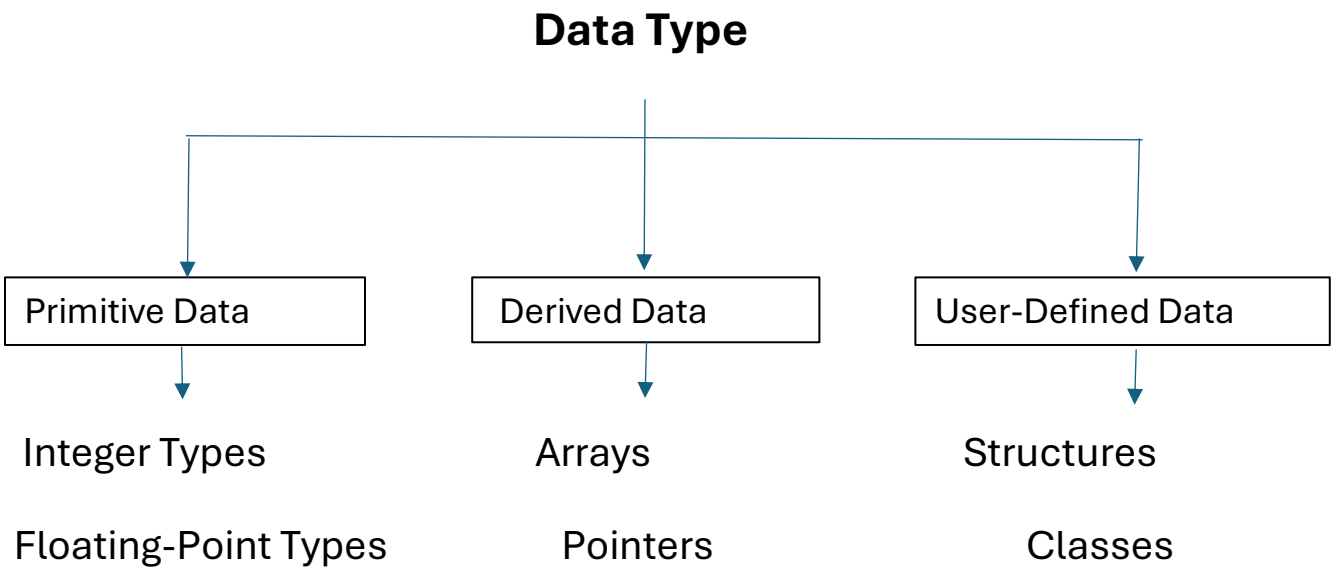
```
#include <iostream>
using namespace std;
main()
{
   // Writing to a file
   ofstream outFile("example.txt");
   outFile << "Hello, File!" << endl;
   outFile.close();

   // Reading from a file
   ifstream inFile("example.txt");
   string content;
   getline(inFile, content);
```

```
    cout << "File Content: " << content << endl;
    inFile.close();
}
```

2. Variables, Data Types, and Operators

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

=> C++ provides a wide range of **data types** to represent different kinds of data in a program. These data types are categorized into three main types: **Primitive Data Types**, **Derived Data Types**, and **User-Defined Data Types**. Here's a detailed explanation of each type, along with examples:



Character Type References Enumerations

Boolean Type

Void Type

1. Primitive Data Types:

Primitive (or fundamental) data types are built into C++ and are the most basic types of data.

a. Integer Types (`int`, `short`, `long`, `long long`)

Example:

```
#include <iostream>

using namespace std;

 main() {

   short a = 100;      // 2-byte integer

   int b = 1000;       // 4-byte integer

   long c = 100000L;   // 'L' indicates long

   long long d = 10000000000LL;  // 'LL' indicates long long

   cout << "Short: " << a << ", Int: " << b << ", Long: " << c << ", Long Long: " << d << endl;

}
```

b. Floating-Point Types (`float`, `double`, `long double`)

`Example`:

```
#include <iostream>

using namespace std;

 main()

{

   float pi = 3.14f;          // 'f' denotes a float literal

   double e = 2.71828;        // Double precision

   long double largeNum = 1.23e30L;  // 'L' denotes long double

   cout << "Float: " << pi << ", Double: " << e << ", Long Double: " <<
largeNum << endl;

}
```

C. Character Type (char):

Example:

```
#include <iostream>

using namespace std;

 main() {

   char grade = 'A';

   cout << "Grade: " << grade << endl;
```

```
}
```

d. Boolean Type (`bool`):

Eample:

```
#include <iostream>

using namespace std;

 main() {

   bool isCPlusPlusFun = true;

   cout << "Is C++ fun? " << isCPlusPlusFun << endl;  // Outputs 1 for true}
```

e. Void Type (`void`):

Example:

```
#include <iostream>

using namespace std;

void greet() {

   cout << "Hello, World!" << endl;

}

main() {

   greet();

}
```

2. Derived Data Types:

Derived data types are built from primitive types. They include:

a. Arrays:

Example:

```
#include <iostream>

using namespace std;

 main() {

   int numbers[5] = {1, 2, 3, 4, 5};  // Array of integers

   cout << "First element: " << numbers[0] << endl;}
```

b. Pointers:

Example:

```
#include <iostream>

using namespace std;

main() {

   int x = 10;

   int* ptr = &x;  // Pointer to the variable x

   cout << "Value: " << x << ", Address: " << ptr << endl;

}
```

C. References:

Example:

```
#include <iostream>

using namespace std;

main() {

   int x = 10;

   int& ref = x;  // Reference to x

   ref = 20;     // Changes x

   cout << "Value of x: " << x << endl;

}
```

3. User-Defined Data Types:

C++ allows you to define your own data types using

a. Structures (`struct`):

Example:

```
#include <iostream>

using namespace std;

struct Student {

   string name;

   int age;

};
```

```
main() {
   Student s1 = {"John", 20};
   cout << "Name: " << s1.name << ", Age: " << s1.age << endl;
}
```

b. Classes:

Exampe:

```
#include <iostream>
using namespace std;
class Car {
public:
   string brand;
   int year;

   void display() {
      cout << "Brand: " << brand << ", Year: " << year << endl;
   }
};
main() {
   Car car1 = {"Toyota", 2023};
```

```
  car1.display();
}
```

C. Enumerations (enum):

Example:

```
#include <iostream>
using namespace std;
enum Color {RED, GREEN, BLUE};
main() {
  Color favoriteColor = GREEN;
  cout << "Favorite Color (integer): " << favoriteColor << endl;  // Outputs 1
}
```

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

=> In C++, **type conversion** refers to converting a value from one data type to another. There are two main types of type conversion:

1. **Implicit Type Conversion** (Automatic or Type Promotion)
2. **Explicit Type Conversion** (Type Casting)

1. Implicit Type Conversion:
Also known as **automatic type conversion** or **type promotion**.

The compiler automatically converts a value from one data type to another without programmer intervention.
It occurs when:

- Assigning a smaller data type to a larger data type (e.g., `int` to `float`).
- Performing operations where operands are of different data types.

There is **no data loss** if the conversion is from a smaller to a larger data type (widening conversion). However, converting from a larger to a smaller type may cause **data truncation.**

Example:

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

class cricket
{
private:

string playername;
int totalmatches;
int averageruns;

public :
  cricket()
  {

  }
        cricket(string pname, int tmatch, int arun)
```

```
  {
     playername=pname;
     totalmatches=tmatch;
     averageruns=arun;
  }
  void print()
  {
       cout<<"\n\n\t enter the player name:"<<playername;
       cout<<"\n\n\t total matches:"<<totalmatches;
       cout<<"\n\n\t averageruns:"<<averageruns;

}
};

main()
{
cricket C1("virat", 285, 72);
cricket C2;

cout<<"\n\n\t object = C1";
cout<<"\n\n\t ----------------------";
C1.print();

C2=C1;
cout<<"\n\n\t object = C2";
cout<<"\n\n\t ----------------------";
C2.print();
}
```

2. Explicit Type Conversion:

Also known as **type casting**.

The programmer manually converts a value from one data type to another.

Used when implicit conversion does not happen automatically, or when precise control over conversion is needed.

Example:

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

class cricket
{
private:
string playername;
int totalmatches;
int averageruns;

public :
   cricket()
   {

   }
cricket(string pname, int tmatch, int arun)
   {
        playername=pname;
        totalmatches=tmatch;
      averageruns=arun;
```

```
  }

  cricket(cricket &C) //E=E1 //copy const
  {
playername=C.playername;
  totalmatches=C.totalmatches;
  averageruns=C.averageruns;
      }
  void print()
  {
      cout<<"\n\n\t enter the player name:"<<playername;
      cout<<"\n\n\t total matches:"<<totalmatches;
      cout<<"\n\n\t averageruns:"<<averageruns;

   }
};
main()
{
cricket C1("virat", 285, 72);
cricket C2;
cout<<"\n\n\t object = C1";
cout<<"\n\n\t ----------------------";
C1.print();

C2=C1;
cout<<"\n\n\t object = C2";
cout<<"\n\n\t ----------------------";
C2.print();
```

}

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

=> In C++, **operators** are symbols or keywords used to perform operations on variables and values. C++ provides a variety of operators that can be classified into different categories.

1. Arithmetic Operators:

These operators perform basic arithmetic operations.

Operator	Description	Example
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	a*b
/	Division	a/b
%	Modulus (Remainder)	a%b

2. Relational (Comparison) Operators:

These operators compare values and return a boolean (`true` or `false`).

Operator	Description	Example
==	Equal to	a==b
!=	Not equal to	a!=b
>	Greater than	a>b
<	Less than	a<b
=>	Greater than or equal	a=>b
<=	Less than or equal	a<=b

3. Logical Operators:

Logical operators are used to perform logical operations, often in conditions.

Operator	Description	Example
&&	Logical AND	(a > b) && (b > c)

`	Logical OR	`
!	Logical NOT	!(a > b)

4. Bitwise Operators:

These operators perform operations at the **bit level**.

Operator	Description	Example
&	Bitwise AND	a&b
`	Bitwise OR	`
^	Bitwise XOR	a^b
~	Bitwise NOT	~a
<<	Left shift	a<<1
>>	Right shift	a>>1

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

=> In C++, **constants** and **literals** are used to represent fixed values that cannot be modified during the program's execution. They provide clarity, safety, and maintainability to code.

1. Constants:

Constants are variables whose values **cannot change** after being defined. They are declared using the `const` keyword or `#define` preprocessor directive.

Purpose of Constants:

- Prevent accidental modification of values.
- Improve code readability (e.g., using `PI` instead of `3.14159`).
- Make code easier to maintain (changing the value of a constant requires a single change).

2.Literals:

Literals are **fixed values** directly used in the code without a variable or constant. They are directly assigned to variables.

Purpose of Constants:

- **Represent Constant Values**: Literals are used to directly specify values such as numbers, text, or logical values.
- **Simplify Code**: They allow you to write values directly into your program instead of pre-defining them as variables.
- **Readability**: Literals make code concise and clear, as they directly express the value being used.
- **Efficiency**: Since literals are constants, they are evaluated at compile time, making the code efficient.

3. Control Flow Statements:

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

=> Conditional statements in C++ allow the program to make decisions and execute specific code based on certain conditions. These statements are used to control the flow of the program by evaluating logical conditions.

1. `if-else` Statement:

The `if-else` statement executes a block of code if a specified condition evaluates to `true`.

If the condition is `false`, the program executes another block of code (if provided).

Key Points:

- The `if` block executes only when the condition is true.

- The `else` block is optional.
- You can use `else if` to check multiple conditions.

2. `switch` Statement:

The `switch` statement evaluates an expression and executes the block of code corresponding to the matching case label.

It is useful for handling multiple conditions based on discrete values.

Key Points:

- The `break` statement is used to exit the `switch` after a case is executed.
- The `default` case is optional but recommended for handling unexpected values.
- The `switch` statement works only with discrete values like integers, `char`, and enumerations (not floating-point values).

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

=>

Feature	**`for` Loop**	**`while` Loop**	**`do-while` Loop**
Initialization	Included in the loop header	Done before the loop separately	Done before the loop separately
Condition Check	Before each iteration	Before each iteration	After the first iteration
Minimum Iterations	Zero	Zero	Zero
Usage	Known number of iterations	Unknown number, condition-driven	At least one iteration needed
Example	Known number of iterationsys	User input validation	Menu-driven applications

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

=> In C++, `break` and `continue` are control statements used within loops to alter their normal flow of execution.

1. `break` Statement:

The `break` statement is used to **terminate the loop immediately**, regardless of the condition.

Once executed, the program exits the loop and continues with the next statement after the loop.

Example:

```
#include <iostream>
using namespace std;
 main()
{
   for (int i = 1; i <= 10; i++)
  {
      if (i == 5)
      {
         break; // Exit the loop when i equals 5
      }

      cout << "Value: " << i << endl;
   }
   cout << "Loop terminated early using break." << endl;
}
```

2. `continue` Statement:

The `continue` statement is used to **skip the rest of the current iteration** and move to the next iteration of the loop.

It does not terminate the loop but forces the loop to check its condition and proceed.

Eample:

```
#include <iostream>
using namespace std;
main()
{
   for (int i = 1; i <= 10; i++)
  {
      if (i % 2 == 0)
      {
         continue; // Skip even numbers
      }
      cout << "Odd Value: " << i << endl;
   }
}
```

4. Explain nested control structures with an example.

=> Nested Control Structures:

Nested control structures occur when one control structure (such as loops, `if` statements, or `switch` statements) is placed inside another.

This allows for more complex decision-making and iterative processes within a program.

Example of Nested Loops:

Problem: Print a multiplication table up to 5x5.

```
#include <iostream>
using namespace std;
main()
{
   for (int i = 1; i <= 5; i++)
  {
      for (int j = 1; j <= 5; j++)
     {
        cout << i * j << "\t"; // Print product of i and j
     }
     cout << endl; // Move to the next row
   }

}
```

4. Functions and Scope:

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

=> A function in C++ is a reusable block of code that performs a specific task. It takes input, processes it, and can return a result.

Functions help in breaking down a program into smaller, manageable, and reusable pieces of code, improving readability and maintainability.

1.Function Declaration (Prototype):

- Specifies the function name, return type, and parameters without providing the actual implementation.
- It tells the compiler about the function's existence and how it should be used.

2.Function Definition:
Provides the actual implementation of the function, defining what the function does.

3.Function Call:
Executes the function by providing the required arguments. The control is transferred to the function definition, and the result (if any) is returned.

2. What is the scope of variables in C++? Differentiate between local and global scope.
=> The scope of a variable in C++ refers to the region in the code where the variable is accessible or visible. Based on where a variable is declared, its scope can be classified as **local** or **global**.
1. Local Scope:

- A variable declared inside a function, block, or loop has a **local scope**.
- It is accessible only within the block in which it is declared.
- Once the block ends, the variable is destroyed, and its memory is deallocated.
- Local variables are not known to other parts of the program.

2. Global Scope:

- A variable declared outside of all functions has a **global scope**.

- It is accessible throughout the program, including all functions and blocks.
- Global variables persist for the lifetime of the program.

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

=> Recursion in C++ is a technique where a function calls itself either directly or indirectly to solve a problem. A recursive function generally has:

1. **Base Case:** The condition where the recursion stops. Without a base case, the function would keep calling itself indefinitely, leading to a stack overflow.
2. **Recursive Case:** The part where the function calls itself to solve a smaller or simpler problem.

How Recursion Works:

Recursion breaks down a problem into smaller instances of the same problem.

The function keeps calling itself with simpler inputs until it reaches the base case, then the results are combined as the stack unwinds.

Example:

The factorial of a non-negative integer nnn (denoted as $n!n!$n!) is the product of all positive integers from 1to n. for Example :

5!=5 × 4 × 3 × 2 × 1=120.

Recursive Formula:

N!=n × (n-1)!

0!=1(Base Case)

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

=> A **function prototype** in C++ is a declaration of a function that specifies its name, return type, and parameters (without providing the body/definition of the function).
It informs the compiler about the function's existence before its actual implementation appears in the program.

Allows Function Usage Before Definition:

- Prototypes enable the use of functions (calls) before their actual definitions appear in the code. This is particularly important in C++ programs where the `main()` function might need to call functions defined later in the file.

Enables Compilation:

- During compilation, the compiler ensures that the function calls match the declared prototypes in terms of the return type and parameter types.

Improves Code Modularity:

- Prototypes allow separating function declarations (e.g., in a header file) from their definitions (e.g., in a source file), enhancing modularity and reusability.

Type Checking:

- The compiler checks the types and number of arguments in the function call against the prototype to prevent mismatches.

5. Arrays and Strings:

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

=> An **array** in C++ is a collection of elements of the same data type stored in contiguous memory locations. It allows storing and accessing multiple values using a single variable name and an index.

Key Features of Arrays:

1. **Fixed Size**: The size of an array is fixed during its declaration.
2. **Homogeneous Elements**: All elements in an array are of the same data type.
3. **Zero-Based Indexing**: The first element is accessed with index `0`, the second with `1`, and so on.

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	A single row or sequence of elements	A grid-like structure with rows and columns
Indexing	Requires one index (e.g., `arr[i]`)	Requires multiple indices (e.g., `arr[i][j]`)
Storage	Stores data linearly in memory	Stores data in a tabular format
Usage	Useful for storing linear lists (e.g., marks)	Useful for tabular data (e.g., matrices)
Complexity	Simpler to use and manage	More complex due to multiple dimensions

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

=> Strings in C++ are used to handle and manipulate text. C++ provides two main ways to work with strings:

1.C-Style Strings: Character arrays terminated with a null character (`'\0'`).

2.C++ Strings: Using the `std::string` class from the Standard Template Library (STL), which provides more functionality and ease of use.

1.C-Style Strings:

A C-style string is essentially a character array terminated with the null character (`'\0'`).

Example:

```
#include <iostream>

using namespace std;

main() {

   char str1[20] = "Hello";

   char str2[20] = "World";

    strcat(str1, str2);

   cout << "Concatenated String: " << str1 << endl;

    cout << "Length of str1: " << strlen(str1) << endl;

    strcpy(str2, "C++ Programming");

   cout << "str2 after copy: " << str2 << endl;

}
```

2.C++ Strings:

The `std::string` class provides a more modern and flexible way to work with strings. It includes many built-in functions for string manipulation.

Example:

```
#include <iostream>

using namespace std;

main() {

   string str1 = "Hello";

   string str2 = "World";

    string str3 = str1 + " " + str2;

   cout << "Concatenated String: " << str3 << endl;

    cout << "Length of str3: " << str3.length() << endl;

    cout << "First character of str3: " << str3[0] << endl;

    cout << "Substring (Hello): " << str3.substr(0, 5) << endl;

    cout << "Position of 'World': " << str3.find("World") << endl;

 }
```

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

=> In C++, arrays are initialized by specifying their type, size, and optionally providing values for the elements. Here's how to initialize one-dimensional (1D) and two-dimensional (2D) arrays:

1D Array Initialization :

A 1D array is a linear sequence of elements of the same type.

Example:

```
int* arr = new int[5];

delete[] arr;
```

2D Array Initialization :

A 2D array is essentially an array of arrays.

```
int** arr = new int*[2];

for (int i = 0; i < 2; ++i)

{

   arr[i] = new int[3];

}

for (int i = 0; i < 2; ++i)

{

   delete[] arr[i];

}
```

delete[] arr;

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

=> In C++, strings can be handled using either C-style strings (null-terminated character arrays) or the more modern and versatile `std::string` class from the Standard Template Library (STL). Here's an overview of string operations and functions for both:

Feature	**C-Style Strings**	**std::string**
Ease of Use	Low	High
Safety	Prone to bugs	Safer
Memory Management	Manual (e.g., new)	Automatic
Functionality	Limited	Extensive

6. Introduction to Object-Oriented Programming:

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

=> Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which encapsulate data and behavior into a single entity.

OOP emphasizes modularity, code reuse, and the modeling of real-world entities. The key concepts of OOP are:

1. Classes and Objects:

- **Class**: A blueprint for creating objects, defining their properties (data members) and behaviors (member functions).
- **Object**: An instance of a class; represents a real-world entity.

Example:

#include <iostream>

```
using namespace std;

class Car
{
public:
   string brand;    int speed;

   void drive()
  {
      cout << brand << " is driving at " << speed << " km/h" << endl;
   }
};



main()
{
   Car car1;
   car1.brand = "Toyota";
   car1.speed = 120;
   car1.drive();
}
```

2. Encapsulation:

Encapsulation is the bundling of data and methods that operate on the data within a single class, and restricting direct access to some of the class's components. This is achieved using **access specifiers**:

- **`public`**: Accessible from outside the class.

- **`private`**: Accessible only within the class.
- **`protected`**: Accessible within the class and its derived classes.

Example:

```
class BankAccount
{
private:
   double balance;
public:
   void setBalance(double b)
  {
      balance = b;
   }
   double getBalance()
  {
      return balance;
   }
};
```

3. Inheritance:

Inheritance allows a class (child) to acquire properties and behaviors of another class (parent). This promotes code reuse and establishes a hierarchy.

- **Base class**: The parent class.
- **Derived class**: The child class

Example:

```
class Father
{
public:
```

```
  void traits()
 {
      cout << "Tall" << endl;
 }
};

class Mother
  {
  public:
       void traits() { cout << "Fair-skinned" << endl;
  }
};

class Child : public Father, public Mother {};
```

4. Polymorphism:

Polymorphism allows one interface to be used for different data types or classes. It enables objects of different types to be treated uniformly.

Example:

```
class Animal
{
  public:
         virtual void sound()
        {
             cout << "Animal sound" << endl;
        }
```

```
};

class Dog : public Animal
{
   public:
         void sound() override
        {
              cout << "Woof!" << endl;
        }
};

void makeSound(Animal* animal)
{
   animal->sound(); // Calls the appropriate function at runtime
}
```

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

=> In C++, **classes** and **objects** are the fundamental building blocks of Object-Oriented Programming (OOP). They allow you to model real-world entities in your code.

Class:

- A **class** is a blueprint or template that defines the properties (data members) and behaviors (member functions) of an object.
- It does not allocate memory until an object of the class is created.

Object:

- An **object** is an instance of a class. When a class is instantiated, it creates an object.
- Each object has its own copy of the class's data members.

Example:

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

class Car
{
  public:
        string brand;
        int speed;

  void drive()
  {
     cout << "The " << brand << " is driving at " << speed << " km/h."
<< endl;
  }
};

main()
{
  Car car1;

  car1.brand = "Toyota";
  car1.speed = 120;

  car1.drive();
```

```
  Car car2;
  car2.brand = "Honda";
  car2.speed = 100;
  car2.drive();
}
```

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

=> Inheritance is a key feature of Object-Oriented Programming (OOP) that allows a class (called the **derived class**) to inherit properties and behaviors from another class (called the **base class**). It promotes code reuse, modularity, and extensibility.

Types of Inheritance in C++:

1.Single Inheritance: A derived class inherits from one base class.

2.Multiple Inheritance: A derived class inherits from more than one base class.

3.Multilevel Inheritance: A derived class inherits from a class, which itself is derived from another class.

4.Hierarchical Inheritance: Multiple derived classes inherit from the same base class.

5.Hybrid Inheritance: A combination of two or more types of inheritance.

Example: Single Inheritance:

```
//public derivation

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

```
class Tops_Student
{
protected :
int id;
string sname;

public :
void get_student()
{
cout<<"\n\n\t Enter Id : ";
cin>>id;
cout<<"\n\n\t Enter Name of the Student : ";
cin>>sname;
}
};

class Tops_Course : public Tops_Student
{
string course;
int duration;

public :
void get_course()
{
cout<<"\n\n\t Enter Course Selected : ";
cin>>course;
cout<<"\n\n\t Enter Duration in months : ";
cin>>duration;
```

```
}

void print_course()
{
cout<<"\n\n\t Course Selected : "<<course;
cout<<"\n\n\t Duration in months : "<<duration;
cout<<"\n\n\t Student's Id : "<<id;
cout<<"\n\n\t Name of the Student : "<<sname;
}
};

main()
{
Tops_Course TC;

TC.get_student();
TC.get_course();
TC.print_course();
}
```

Example: Multilevel Inheritance:

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

class Tops_Stud
{
protected :
string sname;

public :
```

```
        void get_stud()
        {
            sname="Aakash";
}
};

class Tops_Course
{
protected:
string cname;
public :
void get_course()
        {
            cname="Python";
}
};

class Tops_Batch : public Tops_Stud, public Tops_Course
{
string bname;
int duration;

public :
void get_batch()
        {
            bname="8 to 9";
duration=9;
}
```

```
void print()
{
cout<<"\n\n\t Name of the Student : "<<sname;
cout<<"\n\n\t Course Selected : "<<cname;
cout<<"\n\n\t Batch Assigned : "<<bname;
cout<<"\n\n\t Duration in Months : "<<duration;
}

};
main()
{
Tops_Batch TB;

TB.get_stud();
TB.get_course();
TB.get_batch();
TB.print();
}
```

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

=> Encapsulation is a fundamental concept in Object-Oriented Programming (OOP) that refers to the bundling of data (variables) and methods (functions) that operate on that data into a single unit, typically a class.

It also involves restricting direct access to some of the object's components to ensure controlled access and maintain integrity.

Key Features of Encapsulation:

1. **Data Hiding**:
    a. Internal details of the object are hidden from the outside world.
    b. Direct access to sensitive data is restricted using access specifiers like `private` or `protected`.
2. **Controlled Access**:
    a. Access to the class's data is provided through public methods (getters and setters).
3. **Improved Security**:
    a. By controlling access, encapsulation protects the object from unintended interference and misuse.

Access Specifiers in c++:

Encapsulation is achieved using access specifiers:

- **`private`**: Members are accessible only within the class.
- **`protected`**: Members are accessible within the class and its derived classes.
- **`public`**: Members are accessible from outside the class.

Advantages of Encapsulation:

1. **Improved Code Maintenance**:
    a. Encapsulation keeps related code together, making it easier to maintain.
2. **Security**:
    a. Sensitive data is protected from unauthorized access.

3. **Flexibility**:
    a. Implementation details can be changed without affecting external code.
4. **Reusability**:
    a. Encapsulated classes are self-contained and can be reused in different applications