Module 3 Introduction to OOPS Programming

1. Introduction to C++

C++ is a powerful, high-performance programming language that's widely used for developing software, games, operating systems, and real-time systems. It combines features of both high-level and low-level languages, offering flexibility and control over hardware.

Key Features of C++

- Compiled Language: Translates code into machine language for fast execution.
- Object-Oriented: Supports classes, objects, inheritance, polymorphism, and encapsulation.
- Low-Level Manipulation: Allows direct memory access using pointers.
- Standard Template Library (STL): Offers reusable classes and functions for data structures and algorithms.
- Portability: Code can run on various platforms with minimal changes.

THEORY EXERCISE:

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 design philosophy, structure, and how they handle data and functions. Here's a breakdown of the major distinctions:

1. Programming Paradigm

- Procedural Programming:
  - Follows a top-down approach.
  - Focuses on functions or procedures to operate on data.
  - Example languages: C, Pascal.
- Object-Oriented Programming (OOP):
  - Follows a bottom-up approach.
  - Focuses on objects, which combine data and functions (methods).
  - Example languages: Java, Python, C++.

2. Structure

- Procedural:
  - Code is organized into procedures or functions.
  - Data and functions are separate.
- OOP:
  - Code is organized into classes and objects.
  - Data and functions are encapsulated together.

3. Data Handling
   - Procedural:
     - Data is generally global and accessible by any function.
     - Less emphasis on data protection.
   - OOP:
     - Data is usually private or protected.
     - Emphasizes data encapsulation to protect it from unintended access.
4. Reusability
   - Procedural:
     - Reusability is achieved through function reuse.
     - Harder to manage as projects grow in complexity.
   - OOP:
     - Promotes reusability through inheritance and polymorphism.
     - Easier to scale and maintain large applications.
5. Key Concepts
   - Procedural:
     - Uses concepts like functions, modularity, and structured programming.
   - OOP:
     - Built on concepts like:
       - Encapsulation
       - Inheritance
       - Polymorphism
       - Abstraction

6. Real-World Modeling

- Procedural:
  - Less intuitive for modeling real-world entities.
  - More suitable for tasks like mathematical calculations or system-level programming.
- OOP:
  - Naturally maps to real-world entities using objects.
  - Ideal for GUI applications, simulations, and complex systems.

Summary Table

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Top-down	Bottom-up
Focus	Functions	Objects
Data Handling	Global/shared	Encapsulated in objects
Reusability	Limited	High (via inheritance/polymorphism)
Real-World Modeling	Less natural	More intuitive
Code Structure	Functions/modules	Classes/objects

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

1. Modularity (through Classes and Objects)

- OOP Advantage: Code is organized into discrete, reusable units called classes. Each class can represent a concept or real-world entity.
- Benefit: Makes complex programs easier to understand, maintain, and debug by isolating functionality.

2. Reusability (via Inheritance)

- OOP Advantage: Classes can inherit properties and methods from other classes using inheritance.
- Benefit: Promotes code reuse, reducing duplication and allowing easier updates across related classes.

3. Data Encapsulation

- OOP Advantage: Encapsulation hides the internal state of an object and only exposes a controlled interface.
- Benefit: Protects object integrity by preventing external code from making unauthorized changes.

4. Abstraction

- OOP Advantage: OOP supports abstraction, allowing the creation of simple interfaces while hiding complex implementation details.
- Benefit: Simplifies usage and development by letting users work with high-level concepts rather than low-level code.

5. Polymorphism

- OOP Advantage: Enables the same method name to behave differently depending on the object (via method overloading or overriding).

- Benefit: Increases flexibility and extensibility of code, allowing for cleaner and more intuitive implementations.

6. Better Code Maintenance

- OOP Advantage: Modular and self-contained design allows for easier debugging, testing, and updating.
- Benefit: Reduces the risk of bugs and improves the ease of ongoing development.

7. Real-World Modeling

- OOP Advantage: Objects can represent real-world entities with attributes (data) and behaviors (methods).
- Benefit: Makes the code more relatable and intuitive, especially for domain-specific applications like simulations or GUIs.

8. Scalability

- OOP Advantage: Easy to extend programs by adding new classes and objects without rewriting existing code.
- Benefit: Supports large-scale software development and team collaboration.

9. Security

- OOP Advantage: Private and protected access specifiers allow fine-grained control over what data/methods are exposed.
- Benefit: Helps build secure and robust systems by controlling how data is accessed or modified.

Summary Table

OOP Advantage	Why It Matters Over POP
arity	Easier code organization and reuse
Reusability	Inheritance promotes DRY (Don’t Repeat Yourself) code
Encapsulation	Protects data and enforces proper usage
Abstraction	Simplifies complex systems
Polymorphism	Increases flexibility and reuse
Maintenance	Easier to test and debug
Real-World Modeling	Intuitive mapping to real entities
Scalability	Easier to grow and extend programs
Security	Limits unintended interference with data

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

1. Install a C++ Compiler

C++ code must be compiled before it can be run. Common compilers include:

- GCC (GNU Compiler Collection) – popular on Linux/macOS.
- Clang – also used on macOS.
- MSVC (Microsoft Visual C++) – used with Visual Studio on Windows.

Windows

- Option 1: Install MinGW or MSYS2 (for GCC)
- Option 2: Install Microsoft Visual Studio
    - Download from: https://visualstudio.microsoft.com
    - Choose the "Desktop development with C++" workload.

macOS

- Install Xcode Command Line Tools (includes clang):
- xcode-select --install

Linux (Ubuntu/Debian)

- Install g++:
- sudo apt update
- sudo apt install build-essential

2. Choose a Code Editor or IDE

You can write C++ code in any text editor, but IDEs and smart editors provide helpful features like auto-completion and debugging.

Recommended IDEs:

- Visual Studio (Windows) – full-featured IDE.
- CLion – cross-platform, powerful (paid, JetBrains).
- Code::Blocks – lightweight and free.
- Dev C++ – simple for beginners.

Recommended Editors:

- Visual Studio Code (VS Code) – lightweight and extensible.
    - Extensions:
        - C/C++ by Microsoft
        - Code Runner (optional)
    - Get it here: https://code.visualstudio.com

3. Configure Your Editor or IDE

If using VS Code:

1. Install the C/C++ extension from Microsoft.
2. Set up a tasks .json and launch .json file for building and debugging.
3. Make sure the C++ compiler (g++, clang, or cl) is in your system's PATH.

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

Main I/O Streams in C++

Operation	Stream	Description
Input	cin	Standard input (keyboard)
Output	cout	Standard output (screen)
Error	cerr	Standard error (unbuffered)
Log/Error	clog	Standard error (buffered)

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        cout<<"\nHello world";
        getch();
}
```

2. Variables, Data Types, and Operators

THEORY EXERCISE:

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

1.Fundamental (Primitive) Data Types

integer Types

Type	Description	Example
int	Integer numbers	int x = 10;
short	Smaller integer	short s = 100;
long	Larger integer	long l = 100000;
long long	Very large integers	long long ll = 1e18;
unsigned	Only positive values	unsigned int u = 5;

Floating-Point Types

Type	Description	Example
float	Single precision	float pi = 3.14f;
double	Double precision	double d = 3.14159;
long double	Higher precision	long double ld = 3.14L;

Character and Boolean Types

Type	Description	Example
char	Single character	char ch = 'A';
bool	Boolean true/false	bool flag = true;

2. Derived Data Types

Type	Description	Example
Arrays	Collection of same type	int arr[5] = {1,2,3,4,5};
Pointers	Stores address of a variable	int* ptr = &x;
References	Another name for a variable	int& ref = x;
Functions	Blocks of reusable code	int add(int a, int b)

3. User-Defined Data Types

Type	Description	Example
struct	Group of related variables	struct Point { int x, y; };
class	Encapsulation of data & methods	class Car { public: int speed; };
union	Shared memory for multiple vars	union Data { int i; float f; };
enum	Named integer constants	enum Color { RED, GREEN, BLUE };
typedef / using	Alias for data types	typedef unsigned int uint;

Examples in Code

```
#include<iostream.h>
#include<conio.h>
void main()
{
        char name[60];
        int j,c,cpp,rno,t;
        double per;
        clrscr();
        cout<<"\nEnter a student name:";
        cin>>name;
        cout<<"\nEnter a student roll number: ";
        cin>>rno;
        cout<<"\nEnter a java marks: ";
        cin>>j;
        cout<<"\nEnter a c marks: ";
        cin>>c;
        cout<<"\nEnter a c++: ";
        cin>>cpp;
        t=j+c+cpp;
        cout<<"\n\nTotal marks: "<<t;
        per=t/3;
        cout<<"\nPercentage: "<<per;
        if(per>=85)
        {
```

```
 cout<<"\n\nDistinction";
}
else if(per>=75)
{
 cout<<"\n\nFirst class";
}
else if(per>=65)
{
 cout<<"\n\nSecond class";
}
else if(per>=50)
{
 cout<<"\n\nPass class";
}
else
{
 cout<<"\n\nFail";
}
getch();
}
```

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

1. Implicit Type Conversion (Type Promotion)

C++ automatically converts data types when necessary without explicit instruction from the programmer.

expressions with mixed data types.

- When assigning a value of one type to a variable of another type.

Example:

int a = 10;

double b = a; // int is automatically converted to double

Summary Table

Feature	Implicit Conversion	Explicit Conversion
Triggered by	Compiler automatically	Programmer manually specifies
Syntax	No special syntax	(type)variable or static_cast<>
Safety	Generally safe	Can be risky (possible data loss)
Use case	Mixed-type operations	Truncating values, forcing a type
Example	double x = 5;	int y = (int)3.14;

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

1. Arithmetic Operators

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

Example:

```
int a = 10, b = 3;
cout << a + b ;
cout << a % b ;
```

2. Relational (Comparison) Operators

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

Example:

```
if (a > b)
{
   cout << "a is greater than b";
}
```

3. Logical Operators

Operator	Description	Example
&&	Logical AND	a > 0 && b > 0
!	Logical NOT (negation)	!(a > b)

Example:

```
if (a > 0 && b > 0)
{
   cout << "Both are positive";
}
```

4. Assignment Operators

Operator	Description	Example
=	Assign	a = 5
+=	Add and assign	a += 3
-=	Subtract and assign	a -= 2
*=	Multiply and assign	a *= 4
/=	Divide and assign	a /= 2

%=	Modulus and assign	a %= 3

5. Increment and Decrement Operators

Operator	Description	Example
++	Increment	++a; a++
--	Decrement	--a; a--

6. Bitwise Operators

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

7. Conditional (Ternary) Operator

Syntax:

condition ? expression1 : expression2;

Example:

int a = 5, b = 10;

int max = (a > b) ? a : b;

	Description	Example
.	Access member (object)	object.member
->	Access via pointer	pointer->member

Summary Table

Category	Example Operators
Arithmetic	+, -, *, /, %
Relational	==, !=, >, <
Logical	&&, `
Assignment	=, +=, -=
Increment/Decrement	++, --
Bitwise	&, `
Conditional	? :
Sizeof	sizeof(type)
Type Casting	(int), static_cast<>
Member Access	., ->

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

1. Constants

- A constant is a variable whose value cannot be changed after it is initialized.
- Used to define fixed values that should not be modified during program execution.

Why Use Constants?

- Prevents accidental changes to critical values.
- Makes code easier to read and maintain.
- Improves safety and intent clarity.

2. Literals

- A literal is a fixed value written directly in the source code.
- Represents numbers, characters, strings, or booleans.

Types of Literals:

Type	Example	Description
Integer	42, 0, -7	Whole numbers
Floating-point	3.14, -0.5	Decimal numbers
Character	'A', '9'	Single characters (in single quotes)
String	"Hello"	Sequence of characters (double quotes)
Boolean	true, false	Logical true/false values

3. Control Flow Statements

THEORY EXERCISE:

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

Conditional statements in C++ are used to control the flow of a program by executing certain blocks of code only when specific conditions are met. They allow a program to make decisions based on logical conditions.

if, else if, else Statements

Syntax:

```
if (condition) {
   // Executes if condition is true
} else if (another_condition) {
   // Executes if this condition is true
} else {
   // Executes if none of the above are true
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int a,b,c;
```

```
clrscr();
cout<<"\nEnter A:";
cin>>a;
cout<<"\nEnter B:";
cin>>b;
cout<<"\nEnter C:";
cin>>c;
if(a<b)
{
	if(a<c)
	{
		cout<<"\nA is max";
	}
	else
	{
		cout<<"\nB is max";
	}
}
else if(b>a)
{
	cout<<"\nb is max";
}
else
{
	cout<<"\nC is max";
```

```
        }
        getch();
}
```

Switch Statement

Used to compare a variable against multiple constant values. It is often cleaner than multiple if-else statements for comparing a single variable.

Syntax:

```
switch (expression) {
    case constant1:
        // Code block
        break;
    case constant2:
        // Code block
        break;
    default:
        // Code block if no case matches
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
```

```
{
    int a,b,c;
    clrscr();
    cout<<"\nEnter A:";
    cin>>a;
    cout<<"\nEnter B:";
    cin>>b;
    cout<<"\nEnter C:";
    cin>>c;
    if(a<b)
    {
        if(a<c)
        {
            cout<<"\nA is max";
        }
        else
        {
            cout<<"\nB is max";
        }
    }
    else if(b>a)
    {
        cout<<"\nb is max";
    }
    else
```

```
    {
        cout<<"\nC is max";
    }
    getch();
}
```

Differention Between if-else and switch

Feature	if-else	switch
Conditions	Works with any condition	Works only with integral types (int, char, enum)
Flexibility	Supports ranges, expressions	Only supports exact matches
Readability	Can get messy with many conditions	Cleaner for multiple constants
Speed	May be slightly slower	Can be faster with compiler optimization

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

1. for Loop

Syntax:

```
for (initialization; condition; update)
{
   // Loop body
}
```

Example:

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

2. while Loop

Syntax:

```
while (condition)
{
   // Loop body
}
```

Example:

```
int i = 1;
while (i <= 5)
 {
    cout << i << " ";
    i++;
}
// Output: 1 2 3 4 5
```

3. do-while Loop

Syntax:

```
do
{
    // Loop body
} while (condition);
```

Example:

```
int i = 1;
do
{
    cout << i << " ";
    i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

Key Differences Table

Feature	for Loop	while Loop	do-while Loop
Condition check	Before each iteration	Before each iteration	After each iteration
Runs at least once?	✗ No	✗ No	✓ Yes
Best for	Known number of iterations	Unknown number of iterations	Must run at least once
Syntax style	Compact (init, test, update)	Separate init & update	Similar to while, ends with ;

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

1. break Statement

Purpose:

The break statement is used to exit a loop immediately, before the loop condition becomes false.

Use Case:

- To terminate a loop early when a certain condition is met.
- Often used in for, while, or do-while loops and switch statements.

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    //code in c++ language on break statement:
    for(i=0;i<=10;i++)
    {
        if(i==5)
        {
            break;
        }
        else
```

```
        {
            cout<<"\nI: "<<i;
        }
    }
    getch();
}
```

2. continue Statement

Purpose:

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

Use Case:

- When you want to ignore certain values or skip some part of the loop body based on a condition.

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    //code in c++ language on continue statement:
    for(i=0;i<=10;i++)
    {
        if(i==5)
        {
```

```
                continue;
        }
        else
        {
                cout<<"\nI: "<<i;
        }
    }
    getch();
}
```

Summary Table

Statement	Behavior	Effect
break	Exits the loop immediately	Stops the loop completely
continue	Skips the current iteration	Moves to the next loop iteration

4. Explain nested control structures with an example.

Nested control structures are control structures (like if, for, while, switch, etc.) placed inside other control structures.

They allow you to build complex decision-making and looping logic by combining multiple levels of control.

Types of Nested Control Structures

- if inside another if → Nested if
- for loop inside another for loop → Nested for loop
- if inside a for, or for inside an if → Mixed nesting

Example : Nested if Statement

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a,b,c;
    clrscr();
    cout<<"\nEnter A:";
    cin>>a;
    cout<<"\nEnter B:";
    cin>>b;
    cout<<"\nEnter C:";
    cin>>c;
```

```
        if(a<b)
        {
                if(a<c)
                {
                        cout<<"\nA is max";
                }
                else
                {
                        cout<<"\nB is max";
                }
        }
        else if(b>a)
        {
                cout<<"\nb is max";
        }
        else
        {
                cout<<"\nC is max";
        }
        getch();
}
```

Summary

Concept	Example	Use Case
Nested if	if (condition) { if (another) {...}}	Complex decisions
Nested loops	for (...) { for (...) {...} }	Tables, patterns, multi-dimensional data
Mixed nesting (if + loop)	for (...) { if (...) {...} }	Filtering during iteration

4. Functions and Scope

THEORY EXERCISE:

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

function in C++ is a block of code that performs a specific task and can be reused multiple times in a program. Functions help make code modular, easier to read, and maintainable.

Benefits of Using Functions

- Avoids code repetition
- Improves readability
- Makes debugging easier
- Supports modular programming

Types of Functions

- Built-in functions (e.g., cin, cout, sqrt())
- User-defined functions (functions you create)

Key Concepts of Functions

1. Function Declaration (Prototype)

Tells the compiler about the function name, return type, and parameters before its use.

2. Function Definition

Provides the actual body (implementation) of the function.

3. Function Call

Executes the function and passes arguments to it.

Summary Table

Term	Description	Example
Declaration	Tells compiler the function signature	int add(int, int);
Definition	Contains the actual logic	int add(int a, int b) { ... }
Calling	Executes the function with given arguments	add(5, 3);

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

In C++, the scope of a variable refers to the region of the program where the variable can be accessed or modified. It determines the lifetime and visibility of the variable.

Types of Variable Scope

A. Local Scope:

A variable declared inside a block (like a function, loop, or if statement) is called a local variable.

Characteristics:

- Exists only within the block it is defined.
- Cannot be accessed outside that block.
- Memory is allocated when the block starts and freed when it ends.

B. Global Scope

A variable declared outside of all functions, usually at the top of a program, is called a global variable.

Characteristics:

- Accessible from any function in the same file.
- Lifetime lasts from the start to the end of the program.

Local vs Global Scope

Feature	Local Variable	Global Variable
Declared in	Inside a function or block	Outside all functions
Accessible in	Only within the defining block	Any function in the file
Lifetime	Exists during function/block call	Exists throughout program execution
Memory use	Created/destroyed multiple times	Created once and exists globally
Best use case	Temporary data needed only locally	Shared config or state across functions
Risk	Less chance of bugs	Can cause unexpected changes if not handled carefully

Summary

Scope Type	Declared In	Accessible Where	Lifetime
Local	Inside function/block	Only in that function/block	Created and destroyed per call
Global	Outside all functions	Anywhere in the program	Entire program execution
Block	Inside {} braces	Only inside those braces	Until block ends

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

Recursion is a programming technique where a function calls itself directly or indirectly to solve a problem. Each recursive call works on a smaller subproblem, and the process continues until a base condition is met to stop the recursion.

Key Parts of Recursion

1. Base Case – The condition under which the recursion stops.
2. Recursive Case – The part where the function calls itself with a modified argument.

Example:

```
#include<iostream.h>
#include<conio.h>
int factorial(int);
int main()
{
        int n,result;
        clrscr();
        cout<<"\nEnter non nagetive no.:";
        cin>>n;
        result=factorial(n);
        cout<<"\nFactorial of "<<n<<" = "<<result;
        getch();
}
int factorial(int n)
```

```
{
    if(n>1)
    {
        return n*factorial(n-1);
    }
    else
    {
        return 1;
    }
}
```

Summary

Term	Description
Recursion	A function calling itself to solve a problem
Base Case	Stops recursion
Recursive Case	The function keeps calling itself
Use Cases	Factorial, Fibonacci, Tree Traversal, Backtracking

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

function prototype in C++ is a declaration of a function that tells the compiler about the function's name, return type, and parameters — before the function is defined or called.

Why Are Function Prototypes Used?

Reason	Explanation
Forward Declaration	Allows calling a function before it's defined in the file.
Type Checking	Ensures that correct number and types of arguments are passed to the function.
Improves Code Organization	Helps in modular programming, especially with separate files.

Summary

Feature	Description
Function Prototype	Tells the compiler about a function before use
Purpose	Enables early function calls, ensures correctness
Format	return_type function_name(parameter_list);
Required When	Function is defined after main() or in another file

5.Arrays and Strings

THEORY EXERCISE:

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. Arrays allow you to store and manipulate multiple values under a single name, using an index to access individual elements.

Single-Dimensional Arrays

A single-dimensional array (also called a 1D array) stores a list of elements in a single row (or column). It is essentially a linear list.

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a,b[5];
    clrscr();
    for(a=0;a<5;a++)
    {
        cout<<"Enter a "<<a<<" value: ";
        cin>>b[a];
    }
    for(a=0;a<5;a++)
```

```
    {
        cout<<"\nA["<<a<<"] = "<<b[a];
    }
    getch();
}
```

Multi-Dimensional Arrays

A multi-dimensional array is an array of arrays. The most common type is the two-dimensional (2D) array, which is used to represent tabular data (like a matrix).

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int i,j,a[2][2],b[2][2],c[2][2];
    clrscr();
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
            cout<<"Enter row "<<i<<" & column "<<j<<": ";
            cin>>a[i][j];
        }
```

```
}
cout<<"\n\n";
for(i=0;i<2;i++)
{
	for(j=0;j<2;j++)
	{
		cout<<"Enter row "<<i<<" & column "<<j<<": ";
		cin>>b[i][j];
	}
}
for(i=0;i<2;i++)
{
	for(j=0;j<2;j++)
	{
		c[i][j]=a[i][j]+b[i][j];
	}
}
cout<<"\nAddition: ";
for(i=0;i<2;i++)
{
	for(j=0;j<2;j++)
	{
		cout<<"\nA["<<i<<"]["<<j<<"] : "<<c[i][j];
	}
}
```

```
    getch();
}
```

Key Differences:

Feature	Single-Dimensional Array	Multi-Dimensional Array
Structure	Linear (one row or column)	Tabular or grid-like
Syntax Example	int a[5];	int b[3][4];
Element Access	a[2]	b[1][3]
Use Case	Simple lists (e.g., marks, prices)	Tables, matrices (e.g., game board, 2D grid)
Memory Layout	Stored in a single contiguous block	Stored row-wise in a contiguous memory block

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

In C++, strings can be handled using either:

1. C-style strings (character arrays)
2. C++ string class (from the Standard Template Library - STL)

1. C-style Strings

C-style strings are arrays of characters ending with a null character ('\0').

Example:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
        char s1[50],s2[50],s3[50];
        int p,q;
        clrscr();
        cout<<"\nEnter a string:";
        cin>>s1;
        p=strlen(s1);
        cout<<"\nlength of the string:"<<s1;
        cout<<"\nEnter the second string:";
        cin>>s2;
        q=strcmp(s1,s2);
        if(q==0)
```

```
    {
        cout<<"\nBoth string are equal:";
    }
    else
    {
        cout<<"\nBoth string are different:";
    }
    strcat(s1,s2);
    cout<<"\nAfter concat string is:"<<s1;
    strcpy(s3,s1);
    cout<<"\nBefore concat string is:"<<s3;
    strrev(s3);
    cout<<"\nAfter string reversed string:"<<s3;
    getch();
}
```

Common C-style String Functions (from <cstring>):

Function	Description
strlen(str)	Returns length of string
strcpy(dest, src)	Copies one string to another

strcat(dest, src)	Concatenates two strings
strcmp(s1, s2)	Compares two strings

2. C++ string Class

C++ provides a more powerful and flexible way to handle strings through the string class in the <string> header.

Common string Operations:

Operation	Example	Result
Length of string	str.length()	Returns number of characters
Concatenation	str1 + str2	Combines both strings
Substring	str.substr(0, 5)	Extracts a substring
Comparison	str1 == str2	Checks equality
Find a substring	str.find("word")	Returns index or npos
Insertion	str.insert(5, "text")	Inserts text at position
Deletion	str.erase(5, 3)	Deletes part of string

C++ string class is preferred over C-style strings due to:

- Ease of use
- Built-in memory management
- Rich set of functions
- Better safety

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

In C++, arrays can be initialized at the time of declaration using initializers. The syntax and behavior vary slightly depending on whether it is a 1D (one-dimensional) or 2D (two-dimensional) array.

1. One-Dimensional (1D) Arrays

Syntax:

datatype arrayName[size] = {value1, value2, ..., valueN};

Examples:

a. Full Initialization:

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

b. Partial Initialization (remaining values = 0):

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

c. Implicit Size:

int marks[] = {100, 90, 80}; // size = 3

2. Two-Dimensional (2D) Arrays

Syntax:

```
datatype arrayName[rows][columns]
{
    {value1, value2, ...},
    {valueN+1, valueN+2, ...},
    ...
};
```

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

In C++, string manipulation is commonly done using the string class from the Standard Template Library (STL), included via the <string> header. It provides powerful built-in operations and functions to work with text data easily and safely.

Common String Operations

1. Declaration and Initialization

```
#include <string>
string str1 = "Hello";
string str2("World");
string str3; // Empty string
```

2. Input and Output

```
cin >> str1;              // Reads a single word (stops at space)
cout << str1;
```

3. Concatenation

```
string a = "Good", b = "Morning";
string result = a + " " + b;  // "Good Morning"
```

---

4. Length / Size

```
string name = "Alice";
int len = name.length();   // or name.size()
```

---

5. Accessing Characters

```
char first = name[0];     // 'A'
name[2] = 'X';            // Changes 'i' to 'X' → "AlXce"
```

Useful String Functions

Function	Description	Example
length() / size()	Returns number of characters	str.length()
empty()	Checks if string is empty	str.empty() → true/false
append(str)	Adds str at the end	str.append("!")
insert(pos, str)	Inserts str at position pos	str.insert(5, " C++")
erase(pos, len)	Erases len characters from position pos	str.erase(2, 3)
replace(pos, len, str)	Replaces len characters at pos with str	str.replace(0, 5, "Hi")
substr(pos, len)	Returns a substring	str.substr(6, 5)
find(str)	Returns index of first occurrence, or npos	str.find("word")
rfind(str)	Finds last occurrence	str.rfind("a")
compare(str)	Returns 0 if equal, <0 or >0 if not	str1.compare(str2)

Example Program:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
        char s1[50],s2[50],s3[50];
        int p,q;
        clrscr();
        cout<<"\nEnter a string:";
        cin>>s1;
        p=strlen(s1);
        cout<<"\nlength of the string:"<<s1;
        cout<<"\nEnter the second string:";
        cin>>s2;
        q=strcmp(s1,s2);
        if(q==0)
        {
                cout<<"\nBoth string are equal:";
        }
        else
        {
                cout<<"\nBoth string are different:";
        }
        strcat(s1,s2);
```

```
    cout<<"\nAfter concat string is:"<<s1;
    strcpy(s3,s1);
    cout<<"\nBefore concat string is:"<<s3;
    strrev(s3);
    cout<<"\nAfter string reversed string:"<<s3;
    getch();
}
```

Summary

The C++ string class:

- Simplifies string manipulation
- Handles memory automatically
- Offers a wide range of functions for searching, editing, and formatting

6. Introduction to Object-Oriented Programming

THEORY EXERCISE:

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 can contain data (attributes) and functions (methods) that operate on the data. OOP helps in organizing complex programs and promotes code reuse, modularity, and extensibility.

1. Class
   - A class is a blueprint or template for creating objects.
   - It defines attributes (variables) and methods (functions) that the objects will have.
2. Object
   - An object is an instance of a class.
   - It is a real-world entity that has state (data) and behavior (functions).
3. Encapsulation
   - Encapsulation means hiding the internal details of a class and only exposing necessary parts.
   - It protects data from unauthorized access and keeps the code modular.
4. Abstraction
   - Abstraction means showing only the essential features of an object and hiding the complex implementation.

- It reduces complexity and improves efficiency.

5. Inheritance

- Inheritance allows a class (child/derived class) to inherit properties and behaviors from another class (parent/base class).
- Promotes code reuse.

6. Polymorphism

- Polymorphism means “many forms” — the ability to use the same function name or operator for different types or behaviors.
- Two main types:
    - Compile-time (function overloading)
    - Run-time (function overriding using virtual functions)

Summary Table

Concept	Description
Class	Blueprint for creating objects
Object	Instance of a class
Encapsulation	Hiding internal details
Abstraction	Showing essential features only
Inheritance	Reusing code through parent-child classes
Polymorphism	Using the same name with different behaviors

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

Class in C++

A class is a user-defined data type that acts as a blueprint for creating objects. It can contain:

- Data members (variables)
- Member functions (methods)

A class defines the structure and behavior that the object will have.

Object in C++

An object is an instance of a class. It is created using the class and can access the class's members (depending on access specifiers like public, private, etc.).

Example: Class and Object in C++

```
#include<iostream.h>
#include<conio.h>
class Student
{
    int rno;
    char *sname;
    public:
        void getData()
        {
            cout<<"\nEnter a name:";
            cin>>sname;
```

```
                cout<<"\nEnter a roll no.:";
                cin>>rno;
        }
        void putData()
        {
                cout<<"\nStudent Name:"<<sname;
                cout<<"\nRoll number:"<<rno;
        }
};
void main()
{
        Student s1;
        clrscr();
        s1.getData();
        s1.putData();
        getch();
}
```

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

Inheritance is a key feature of Object-Oriented Programming (OOP) in C++ that allows a class (derived/child) to inherit properties and behaviors (data and functions) from another class (base/parent).

- It promotes code reuse, modularity, and hierarchical classification.
- Inheritance models real-world "is-a" relationships.

Example: A Dog *is a* type of Animal.

Types of Inheritance in C++:

1. Single Inheritance (one base, one derived class)
2. Multiple Inheritance (one derived class inherits from multiple base classes)
3. Multilevel Inheritance (derived from a derived class)
4. Hierarchical Inheritance (multiple classes inherit from the same base class)
5. Hybrid Inheritance (combination of two or more types)

Example: Single Inheritance

```
#include<conio.h>
class A
{
    int a;
    public:
    #include<iostream.h>
```

```
        void getA()
                {
                        cout<<"\nEnter value of A: ";
                        cin>>a;
                }
                void putA()
                {
                        cout<<"\nA: "<<a;
                }
};
class B:public A
{
        int b;
        public:
                void getB()
                {
                        cout<<"\nEnter value of B: ";
                        cin>>b;
                }
                void putB()
                {
                        cout<<"\nB: "<<b;
                }
};
void main()
```

```
{
        clrscr();
        B b1;
        b1.getA();
        b1.getB();
        b1.putA();
        b1.putB();
        getch();
}
```

Access Specifiers in Inheritance:

Inheritance Type	Public Members in Base	Protected Members in Base	Private Members in Base
public	Public in Derived	Protected in Derived	Not inherited
protected	Protected in Derived	Protected in Derived	Not inherited
private	Private in Derived	Private in Derived	Not inherited

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

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (variables) and the methods (functions) that operate on that data into a single unit — typically a class — and restricting direct access to some of the object's components.

Why Use Encapsulation?

- To protect data from unauthorized access or accidental modification.
- To enforce a controlled interface through getter/setter functions.
- To maintain modularity and make code easier to understand and maintain.

How is Encapsulation Achieved in C++?

Encapsulation is implemented using access specifiers in a class:

- private – members cannot be accessed from outside the class.
- public – members can be accessed from outside.
- protected – accessible in derived classes (used in inheritance).