**(i) Object Oriented Concepts**

**Object Oriented Programming- a new paradigm**

Paradigm means a style or approach to programming. Earlier, programming followed a procedural paradigm, where the focus was on functions and procedures (e.g., C language).

Object-Oriented Programming (OOP) is a paradigm that focuses on objects — real-world entities represented in code — combining both data attributes and methods.

**Abstraction**

Abstraction is the process of hiding complex implementation details and showing only the necessary features of an object.

**Forms of Abstraction**

1. **Procedural Abstraction:** Where complex operations are encapsulated into a single, named procedure (or function/routine) that can be called without needing to know its internal implementation details, focusing only on its input, output, and intended behavior. Its Achieved through functions/methods.

Example: A **print ()** function hides the internal logic of how printing happens.

1. **Data Abstraction:** Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Its Achieved through classes and objects.

Example: A Bank Account class exposes **deposit ()** and **withdraw ()** methods but hides the actual balance storage logic.

**Classes**

A class is a blueprint/template for creating objects. It defines properties (data members) and methods (functions).

class Car {

public:

string model;

int speed;

void accelerate() {

speed += 10;

}

};

**Objects**

An object is a real instance of a class. You can create multiple objects from one class.

Car myCar; // Object created from Car class

myCar.model = "Tesla";

myCar.speed = 50;

myCar.accelerate();

**Polymorphism**

Poly = many, Morphism = forms → “many forms” Polymorphism allows the same function name or operator to behave differently based on context.

**Types of Polymorphism**

1. **Compile-time (Static):** it a feature where the compiler determines which method to execute at compile time based on the method's signature (name and parameters).

**Method Overloading**: This is the primary way to achieve compile-time polymorphism. You can have several methods with the same name, but they must differ in either the number or the data type of their parameters.

**Example:** A class might have a add method that can take two integers, two floats, or even a string and an integer, with the compiler choosing the correct add method to run based on the arguments provided.

1. **Run-time (Dynamic):** Runtime polymorphism, also known as dynamic polymorphism or late binding, is a concept in object-oriented programming where the decision of which method to execute is made at runtime, not at compile time. It is primarily achieved through method overriding, where a derived class provides a specific implementation for a method that is already defined in its parent class. This requires an inheritance or "IS-A" relationship between the classes.

class Animal {

public:

virtual void speak() { cout << "Animal sound\n"; }

};

class Dog : public Animal {

public:

void speak() override { cout << "Bark\n"; }

};

**Data Encapsulation**

Encapsulation means **wrapping data and functions together** into a single unit (class) and controlling access to data.

class BankAccount {

private:

double balance;

public:

void deposit(double amount) { balance += amount; }

double getBalance() { return balance; }

};

**Data Hiding**

**Data Hiding** is about **restricting direct access** to data members and providing controlled access through methods.

account.balance = 1000; // Not allowed (private)

account.deposit(1000); // Correct way

**Inheritance**

**Inheritance** allows a class (**child**) to **reuse the properties and methods** of another class (**parent**).

**Types of Inheritance (C++):**

1. **Single Inheritance:** One derived class inherits from one base class.
2. **Multiple Inheritance:** One derived class inherits from multiple base classes.
3. **Multilevel Inheritance:** A class is derived from another derived class (forming a chain).
4. **Hierarchical Inheritance:** Multiple derived classes inherit from one base class.
5. **Hybrid Inheritance:** Hybrid Inheritance**.**

|  |  |  |
| --- | --- | --- |
| Type | Description | Example |
| Single | A → B | Dog inherits Animal |
| Multiple | A, B → C | Child inherits Father & Mother |
| Multilevel | A → B → C | Grandfather → Father → Son |
| Hierarchical | A → B, C, D | Vehicle → Car, Bike |
| Hybrid | Mix of above | Multiple + Hierarchical |

**(ii)Basics of C++, Classes and Objects**

**Features of C++**

1. Fast execution (compiled language)
2. Rich standard library
3. Portable and platform-independent
4. Supports procedural + OOP
5. Closer to hardware (low-level memory control with high-level abstraction)

**Tokens**

Tokens are the smallest elements of a C++ program.

Types of Tokens:

1. Keywords
2. Identifiers
3. Constants
4. Strings
5. Operators
6. Punctuators / Symbols

**Keywords**

Reserved words that have special meaning in C++ (cannot be used as identifiers).

Examples:

int, float, if, else, return, class, public, private, void, new, delete, try, catch, etc.

**Data types**

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| int | Integer numbers | 5, -20 |
| float | Floating-point numbers | 5.5 |
| double | Double precision numbers | 3.14159 |
| char | Single character | 'A', '9' |
| bool | Boolean | true, false |
| void | No value | functions |
| wchar\_t | Wide character | L'A' |

**Operators**

**Arithmetic: + - \* / %**

**Relational: == != > < >= <=**

**Logical: && || !**

**Assignment: = += -= \*= /= %=**

**Increment/Decrement: ++ --**

**Conditional / Ternary: condition? x: y**

**Bitwise: & | ^ ~ << >>**

**Pointer operators: \* (value at), & (address of)**

**Scope resolution: ::**

**Manipulators**

Manipulators are used to **format the output**.  
 Defined in <iomanip> and <iostream>.

#include <iostream>

#include <iomanip>

using namespace std;

int main()

{

    int a = 10;

    cout << a << **endl**; // endl is Manipulator

    cout << **setw(5)** << **setfill('\*')** << a << endl;

    return 0;

}

**Console input / Output**

cout → console output

cin → console input

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter age: ";

cin >> age;

cout << "Your age is " << age;

return 0;

}

**Control statements (conditional and loops)**

int main()

{

for (int i = 1; i <= 10; i++)

{

if (i % 2 != 0)

{

cout << i << endl;

}

}

return 0;

}

**Functions, Classes**

class Person {

public:

string name;

int age;

void introduce() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

**Instantiation**

int main() {

Person p1; // Instantiation

p1.name = "Saidur";

p1.age = 20;

p1.introduce();

return 0;

}

**Destructor and Constructor**

class Abc

{

public:

int a;

int b;

int sum()

{

return a + b;

};

Abc() // Constructor

{

a = 1;

b = 2;

}

~Abc() // Destructor

{

a = 0;

b = 0;

}

};

int main()

{

Abc abc1;

abc1.a = 12;

abc1.b = 12;

cout << abc1.sum() << endl;

Abc abc2;

cout << abc2.sum() << endl;

return 0;

}

**Polymorphism**

Polymorphism = “Many forms”

It allows the same function, operator, or method to behave differently depending on the context (object or data type). In Object-Oriented Programming (OOP), polymorphism helps us reuse names and write flexible, extensible code.

2. Types of Polymorphism

1. **Compile-time Polymorphism**

**Operator Overloading (Example of Compile-time Polymorphism)**

#include <iostream>

using namespace std;

class Calculator {

public:

// Add 2 integers

int add(int a, int b) {

return a + b;

}

// Add 3 integers

int add(int a, int b, int c) {

return a + b + c;

}

// Add 2 doubles

double add(double a, double b) {

return a + b;

}

};

int main() {

Calculator calc;

cout << "Sum of 2 ints: " << calc.add(2, 3) << endl;

cout << "Sum of 3 ints: " << calc.add(2, 3, 4) << endl;

cout << "Sum of 2 doubles: " << calc.add(2.5, 3.5) << endl;

return 0;

}

**Function Overloading (Example of Compile-time Polymorphism)**

#include <iostream>

using namespace std;

class Complex {

public:

int real, imag;

Complex(int r = 0, int i = 0) {

real = r;

imag = i;

}

// Overload the + operator

Complex operator + (Complex const &obj) {

Complex result;

result.real = real + obj.real;

result.imag = imag + obj.imag;

return result;

}

};

int main() {

Complex c1(2, 3), c2(4, 5);

Complex c3 = c1 + c2; // uses overloaded +

cout << "Result: " << c3.real << " + " << c3.imag << "i" << endl;

return 0;

}

**2. Runtime Polymorphism**

**Function overriding (in inheritance) is the example of runtime Polymorphisms**

**Virtual functions are the example of runtime Polymorphisms**

#include <iostream>

using namespace std;

class Animal {

public:

virtual void sound() { // virtual enables runtime polymorphism

cout << "Animal makes a sound" << endl;

}

};

class Dog : public Animal {

public:

void sound() override {

cout << "Dog barks 🐶" << endl;

}

};

class Cat : public Animal {

public:

void sound() override {

cout << "Cat meows 🐱" << endl;

}

};

int main() {

Animal\* a; // pointer of base class

Dog d;

Cat c;

a = &d;

a->sound(); // Dog's sound() is called

a = &c;

a->sound(); // Cat's sound() is called

return 0;

}

**1. Basic Meaning**

**Inheritance** is one of the **core pillars of Object-Oriented Programming (OOP)**.  
It allows a **class to reuse** the **properties (data)** and **behaviors (functions)** of **another class**, without rewriting the same code again.

**2. Basic Structure**

class Parent {

// properties and methods

};

class Child : public Parent {

// additional properties and methods

};

* Parent → **Base class** (or Superclass)
* Child → **Derived class** (or Subclass)
* public → Type of inheritance (there are also private and protected)

**3. Why Use Inheritance?**

**Reusability** → Don’t repeat code.  
**Extensibility** → Add or change behavior easily.  
**Organization** → Build hierarchical relationships (e.g., Animal → Dog → German Shepherd).  
**Polymorphism** → Inheritance enables **runtime polymorphism** (virtual functions, overriding, etc.)

**4. Types of Inheritance in C++**

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Single** | One base class → One derived class |
| **Multilevel** | A derived class becomes base for another |
| **Multiple** | A derived class inherits from multiple base classes |
| **Hierarchical** | Multiple derived classes from one base |
| **Hybrid** | Combination of two or more types |

**A. Single Inheritance**

#include <iostream>

using namespace std;

// Base Class

class Person {

public:

string name;

int age;

void introduce() {

cout << "Hi, I am " << name << ", Age " << age << endl;

}

};

// Derived Class

class Student : public Person {

public:

string course;

void study() {

cout << name << " is studying " << course << endl;

}

};

int main() {

Student s;

s.name = "Saidur";

s.age = 21;

s.course = "C++";

s.introduce(); // from base class

s.study(); // from derived class

return 0;

}

**Output**:

Hi, I am Saidur, Age 21

Saidur is studying C++

**B. Multilevel Inheritance**

#include <iostream>

using namespace std;

class LivingBeing {

public:

void breathe() {

cout << "Breathing..." << endl;

}

};

class Animal : public LivingBeing {

public:

void move() {

cout << "Moving like an animal" << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Dog barks 🐶" << endl;

}

};

int main() {

Dog d;

d.breathe(); // from LivingBeing

d.move(); // from Animal

d.bark(); // from Dog

return 0;

}

**C. Multiple Inheritance**

#include <iostream>

using namespace std;

class Teacher {

public:

void teach() {

cout << "Teaching..." << endl;

}

};

class Researcher {

public:

void research() {

cout << "Researching..." << endl;

}

};

class Professor : public Teacher, public Researcher {

public:

void guide() {

cout << "Guiding students..." << endl;

}

};

int main() {

Professor p;

p.teach(); // from Teacher

p.research(); // from Researcher

p.guide(); // from Professor

return 0;

}

**D. Hierarchical Inheritance**

#include <iostream>

using namespace std;

class Vehicle {

public:

void start() {

cout << "Vehicle starting..." << endl;

}

};

class Car : public Vehicle {

public:

void drive() {

cout << "Car driving 🚗" << endl;

}

};

class Bike : public Vehicle {

public:

void ride() {

cout << "Bike riding 🏍️" << endl;

}

};

int main() {

Car c;

Bike b;

c.start(); // from Vehicle

c.drive();

b.start(); // from Vehicle

b.ride();

return 0;

}

One base class → multiple derived classes.

**5. Access Specifiers in Inheritance**

When inheriting, you can use:

* public
* protected
* private

This affects **how members of the base class** are visible in the derived class.

|  |  |  |  |
| --- | --- | --- | --- |
| **Base class member** | **Public inheritance** | **Protected inheritance** | **Private inheritance** |
| public | public in derived | protected in derived | private in derived |
| protected | protected in derived | protected in derived | private in derived |
| private | not inherited | not inherited | not inherited |

**Pointers**

**1. What is a Pointer?**

A **pointer** is a **variable** that stores the **memory address** of another variable. Instead of storing a value directly, it stores **where that value is located in memory**.

**2. Pointer Syntax**

datatype \*pointerName;

* \* → declares a pointer
* & → “address of” operator

Example:

int num = 10;

int \*p; // declare a pointer to int

p = &num; // store the address of num

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

cout << "Address of num: " << &num << endl;

cout << "Pointer p (address stored): " << p << endl;

cout << "Value pointed by p: " << \*p << endl;

**Output example:**

Value of num: 10

Address of num: 0x7ffd8b2c

Pointer p (address stored): 0x7ffd8b2c

Value pointed by p: 10

p stores the address of num.  
\*p accesses the **value** stored at that address (this is called **dereferencing**).

**3. Changing Values via Pointers**

You can **change the original variable** using its pointer:

#include <iostream>

using namespace std;

int main() {

int x = 5;

int \*ptr = &x;

\*ptr = 20; // change value through pointer

cout << "x = " << x << endl; // x = 20

cout << "\*ptr = " << \*ptr << endl; // 20

return 0;

}

Since ptr points to x, changing \*ptr also changes x.

**4. Pointers and Arrays**

In C++, **array name itself is a pointer** to the **first element**.

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

int \*p = arr; // no & needed

cout << \*p << endl; // 10 (first element)

cout << \*(p + 1) << endl; // 20 (second element)

cout << \*(p + 2) << endl; // 30 (third element)

**5. Pointer to Pointer (Double Pointer)**

A **pointer to pointer** stores the **address of another pointer**.

int x = 10;

int \*p = &x;

int \*\*pp = &p; // pointer to pointer

cout << "x = " << x << endl;

cout << "\*p = " << \*p << endl;

cout << "\*\*pp = " << \*\*pp << endl;

pp → address of p  
 \*pp → gives p  
 \*\*pp → gives value of x

**6. Pointers and Functions**

Pointers let you **pass variables by reference** to functions (without using & in the parameter).

Example: **Swap two numbers** using pointers:

#include <iostream>

using namespace std;

void swapNumbers(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x = 5, y = 10;

swapNumbers(&x, &y);

cout << "x = " << x << ", y = " << y << endl; // x = 10, y = 5

return 0;

}

Using pointers allows the function to modify **original variables**, not just copies.

**7. Dynamic Memory Allocation with Pointers**

Using new and delete, you can create variables at runtime:

int \*ptr = new int; // allocate memory for one int

\*ptr = 50;

cout << \*ptr << endl; // 50

delete ptr; // free memory

For arrays:

int \*arr = new int[3]; // dynamic array

arr[0] = 10;

arr[1] = 20;

arr[2] = 30;

cout << arr[1] << endl; // 20

delete[] arr; // free array memory

1. **Analysis of Algorithm**

**Introduction to algorithm design and Data structures**

An algorithm is a finite sequence of well-defined instructions to solve a particular problem.

characteristics of a Good Algorithm

1. Input: Zero or more inputs
2. Output: At least one output
3. Definiteness: Each step must be clearly defined
4. Finiteness: Must terminate after finite steps
5. Effectiveness: Each step must be basic and executable
6. Generality: Should work for all valid inputs of the problem

Comparison of Algorithms, Complexity in terms of **space** and **time**, Calculation of 0- notation. Abstract Data type and its implementation with a Rational number example

1. **Searching and Sorting**

Searching- Linear and Binary Search, Sorting- Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Comparison of various searching and sorting techniques in terms of time complexity

1. **Elementary Data Structures: Arrays, Linked Lists**

Representation of arrays-single and multidimensional, Address calculation using row major ordering, Various operations on arrays, Linked Lists-Singly Linked List, Double linked List, Circular Linked List- traversing, deleting, inserting, searching, counting, reversing, printing of nodes.

1. **Stacks and Queues**

Stack ADT, Implementation of stack using array and linked list, Application of Stack- Evaluation of postfix/prefix expression, Queue ADT, Implementation of queue using Array and Linked List

1. **Trees**

Definition and notations, Binary Search Trees Implementation. Traversals using stacks and recursion - In-order, post-order, pre-order techniques, Threaded binary tree, B-trees with implementation of 2-3 trees.

1. **Graphs**

Definition and notations, Components of Graphs, Types of Graphs, Graph Implementation using Adjacency Matrix and Adjacency List algorithms and programs, Graph Traversal Methods: Depth First Search and Breadth First Search.