

UNIT I — Object Oriented Systems with C++

UNIT I:

Introduction: An Overview of Object Oriented Systems Development, Object Basics, Object Oriented Systems Development Life Cycle.

Object Oriented Methodologies: Rumbaugh Methodology, Booch Methodology, Jacobson Methodology, Patterns, Frameworks, Unified Approach, Unified Modeling Language, Use case, class diagram, Interactive Diagram, Package Diagram, Collaboration Diagram, State Diagram, Activity Diagram.

Object Oriented Analysis: Identifying use cases, Object Analysis, Classification, Identifying Object relationships, Attributes and Methods.

Object Oriented Design: Design axioms, Designing Classes, Access Layer, Object Storage, Object Interoperability.

1) Basics — what OO is and why use it

Object-Oriented Systems Development (OOSD) models real-world things as **objects** (state + behavior).

Why OO (short):

- maps to the problem domain,
- improves modularity & reuse,
- isolates change (encapsulation),
- supports polymorphism & extension.

ATM (very basic): objects = Customer, Card, Account, ATM_UI, Transaction.

2) Core object basics — from beginner → deeper

- **Class:** blueprint (attributes + methods).
- **Object:** runtime instance of a class.
- **Encapsulation:** hide data; expose methods.
 - *ATM example:* Account.balance is private, only accessible via deposit()/withdraw().
- **Abstraction:** expose only needed operations.
 - *ATM UI* shows “Withdraw” but not internals of transaction processing.
- **Inheritance:** subclass reuses superclass.
 - e.g., SavingsAccount : public Account.

- **Polymorphism:** one interface, multiple behaviors.
 - e.g., `Transaction* t = new CashWithdrawal(); t->process();` uses virtual method dispatch.
 - **Association / Aggregation / Composition:**
 - Association: Customer — Account (linked).
 - Aggregation: Bank aggregates Branch (independent).
 - Composition: ATM composed of CashDispenser (lifetime bound).
 - **Message Passing:** objects call methods on each other; sequence diagrams show this.
-

3) OOSD Life Cycle — basic steps and advanced iterative practice

Phases (simple):

1. Requirements / Use cases
2. Object-Oriented Analysis (OOA)
3. Object-Oriented Design (OOD)
4. Implementation (C++, etc.)
5. Testing & Maintenance

Advanced practice: do these iteratively in short cycles (RUP/Agile): capture critical use cases first, build prototypes (UI + mock services), refine objects & APIs, add nonfunctional requirements (security, performance) each iteration.

4) Object-Oriented Methodologies — what to use when

Rumbaugh (OMT)

- **Focus:** analysis → model three views: object (static), dynamic (states/events), functional (data flow).
- **Use when:** heavy analysis is needed (complex domain models).
- *ATM:* model transaction life cycle with dynamic model, object classes with object model.

Booch

- **Focus:** detailed design & implementation guidance, good notations for components.
- **Use when:** designing component interfaces & low-level system architecture.

Jacobson (OOSE)

- **Focus: Use Cases** drive the design — excellent for requirements.
- **Use when:** requirement clarity & user interactions are primary (like banking kiosks).
- **Unified Approach:** combine the best parts of above; use UML as common notation.

Comparison (practical):

- Use **Jacobson** to capture requirements, **Rumbaugh** to analyze structure and states, **Booch** to design components and implementation details — combine them.
-

5) Patterns & Frameworks — basic → applied examples

- **Patterns** are proven solutions to recurring problems.
 - **Singleton** — single instance (e.g., Logger or ATMController).
 - **Factory** — create transaction objects (TransactionFactory::create(type)).
 - **Observer** — UI updates when transaction status changes.
 - **Strategy** — pluggable validation strategies (chip, PIN, biometric).
- **Frameworks:** reusable domain scaffolds (e.g., banking framework supplying TransactionManager, AccountDAO).

Practical tip: In exam/assignment name a pattern and show where it fits (1–2 lines).

6) UML — each diagram from basic → advanced with ATM examples

Use Case Diagram (basic)

- **Purpose:** show actors & system capabilities (requirements).
- **ATM example:** actor Customer — use cases: Withdraw Cash, Deposit, Balance Enquiry.

Class Diagram (static) — more detail

- **Purpose:** classes, attributes, methods, relationships.
- **ATM snippet:**

```
csharp Copy code  
  
Account  
- accNo: string  
- balance: double  
+ getBalance(): double  
+ debit(amount): bool
```

- Include multiplicity (Customer 1..* — Accounts), access (+, -), associations, aggregation/composition.

Sequence Diagram (interaction) — basic→advanced

- **Purpose:** show message order & lifelines.
- **Withdraw example steps:** Customer → ATM_UI → AuthController → BankServer → Account → CashDispenser.
- Advanced: show synchronous vs asynchronous messages, return values, timeouts & error flows.

Collaboration/Communication Diagram

- Emphasizes object links & message numbering. Use when structure matters (e.g., showing that ATM_UI holds a reference to CardReader).

State Diagram (dynamic)

- **Purpose:** object lifecycle.
- **Transaction states:** Initiated → Authorized → Processing → Completed / Failed.
- Advanced: nested states (substates), entry/exit actions, concurrency regions.

Activity Diagram

- **Purpose:** workflow, parallelism.
- **Withdraw:** branch on sufficientBalance?, fork for async logging & dispensing.
- Advanced: swimlanes (who does what — UI, BankServer, ATM hardware).

Package Diagram

- Group classes: UI, BusinessLogic, Persistence, DeviceDrivers.
-

7) Object-Oriented Analysis (OOA) — step-by-step with ATM (practical method)

Goal: get from requirements → candidate classes & relationships.

1. Collect Use Cases

- Example: Withdraw Cash — actor, precondition, main flow, alt flows.

2. Identify candidate objects — underline nouns in use case text.

- card, PIN, amount, account, transaction, dispenser.

3. Classify objects

- **Boundary** (UI, CardReader), **Entity** (Account, Transaction), **Control** (TransactionManager).

4. Define relationships & multiplicities

- Customer 1..* — Account

5. Define attributes & operations

- Account.balance, Account.debit(amount).

6. Refine with CRC cards (Class, Responsibilities, Collaborators).

7. Validate with sequence diagram to ensure interactions are covered.

Advanced OOA: capture constraints (business rules), candidate keys and metadata — e.g., Account.accNo is unique (candidate key), withdrawal limit constraints.

8) Object-Oriented Design (OOD) — principles → concrete application

Design axioms / principles (with ATM examples)

- **Encapsulate what varies.**
 - If fee calculation changes, create FeeCalculator interface and concrete strategies.
- **Program to interfaces, not implementations.**
 - Depend on IAccountRepository not SqlAccountRepository.
- **Low coupling, high cohesion.**
 - Account only manages balance; TransactionManager orchestrates.
- **Single Responsibility Principle.**
 - AuthController handles authentication only.
- **Prefer composition over inheritance.**
 - ATM composes a CardReader rather than inheriting from it.

Designing Classes — checklist & example

- For each class list: responsibilities, attributes, methods, collaborators.
- Example TransactionManager:
 - Responsibilities: validate, coordinate debit, record transaction.
 - Collaborators: AccountDAO, AuthController, CashDispenser.
 - Methods: requestWithdrawal(cardNo, amount).

Interfaces & Abstract Classes

- Use abstract classes where you want common behavior + some unimplemented operations (e.g., Transaction with process() abstract, specialized by CashWithdrawal, BalanceEnquiry).

Error handling & recovery design (advanced)


- Add timeouts, retries, fallback (e.g., if BankServer unavailable, return user-friendly message and queue transaction for later).

9) Access Layer, Object Storage & Persistence — simple → advanced

Access Layer / DAO pattern (basic)

- **Purpose:** separate persistence from business logic.
- Example interface (pseudo-C++):

cpp

 Copy code

```
class IAccountDAO {  
public:  
    virtual Account getAccount(const string& accNo) = 0;  
    virtual void updateAccount(const Account& a) = 0;  
    virtual ~IAccountDAO() {}  
};
```

TransactionManager depends on IAccountDAO (injected), so tests can use a mock DAO.

Object Storage options

- **File-based serialization** (JSON, XML, binary) — simple, good for logs.
- **Relational DB** + ORM mapping — common in enterprise (careful: object-relational impedance).
- **Object DB** — store objects natively (less common).
- **Advanced concerns:** transactions (ACID), locking, optimistic concurrency, lazy loading.

Mapping & constraints

- Map class associations to foreign keys, model multiplicities, set candidate keys (accNo).
- Implement constraints (e.g., withdrawal limit) at DB and application layer.

10) Object Interoperability — simple APIs → distributed systems

Basic interop

- **REST / JSON** APIs for web services (e.g., ATM → BankServer /validatePIN, /transactions).
- **SOAP** / WSDL for strong contract-based web services (legacy enterprises).

Advanced interop (enterprise)

- **Message queues** (RabbitMQ, Kafka) for asynchronous operations (e.g., offline reconciliation).
- **gRPC** for efficient binary RPC between services.
- **Middleware (CORBA/RMI)** older solutions, conceptually similar (remote objects).

Versioning & contracts

- Design stable API contracts, include versioning (/v1/transactions) and backward compatibility strategies.


11) Worked mini-artifacts (copyable for exams)

Use Case — Withdraw Cash

- **Actor:** Customer
- **Preconditions:** Card valid; ATM has cash.
- **Main Flow:** Insert Card → Enter PIN → Select Withdraw → Enter Amount → Verify balance → Dispense Cash → Update Account → Eject Card.
- **Alternate:** insufficient funds → show error.

Class (exam-ready)

php

 Copy code

```
class Account {  
- accNo: string  
- balance: double  
+ getBalance(): double  
+ debit(amount: double): bool  
+ credit(amount: double): void  
}
```


Sequence (textual)

1. Customer -> ATM_UI: insertCard()
2. ATM_UI -> AuthController: validate(card, pin)
3. ATM_UI -> TransactionManager: requestWithdraw(amount)
4. TransactionManager -> IAccountDAO: getAccount(accNo)
5. TransactionManager -> Account: debit(amount)
6. TransactionManager -> CashDispenser: dispense(amount)
7. TransactionManager -> IAccountDAO: updateAccount()

State (Transaction)

- Initiated —(authorize)-> Authorized —(process)-> Processing —> Completed / Failed
-

12) Patterns & where to apply (practical mapping)

- **Factory:** create Transaction objects from type.
 - **Observer:** ATM UI listening to Transaction status events.
 - **Strategy:** different authentication strategies (PIN, biometric).
 - **Repository/DAO:** persistence layer for Account, Transaction.
-

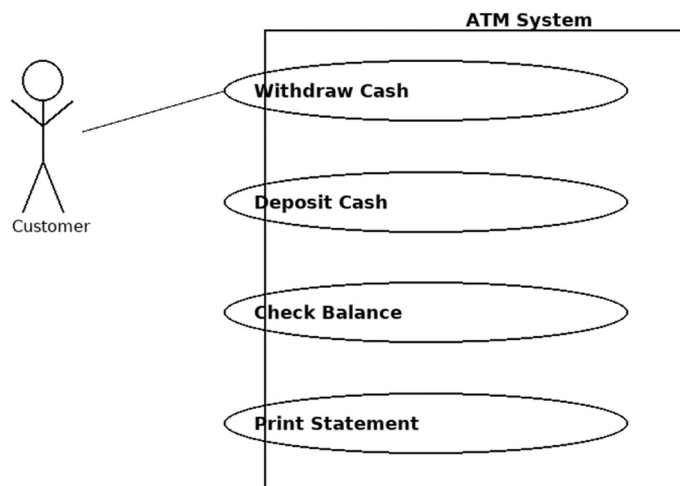
13) Advanced concerns (short notes)

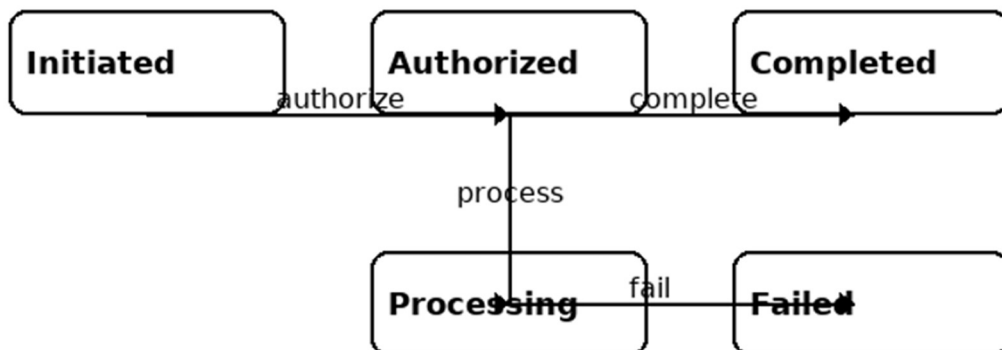
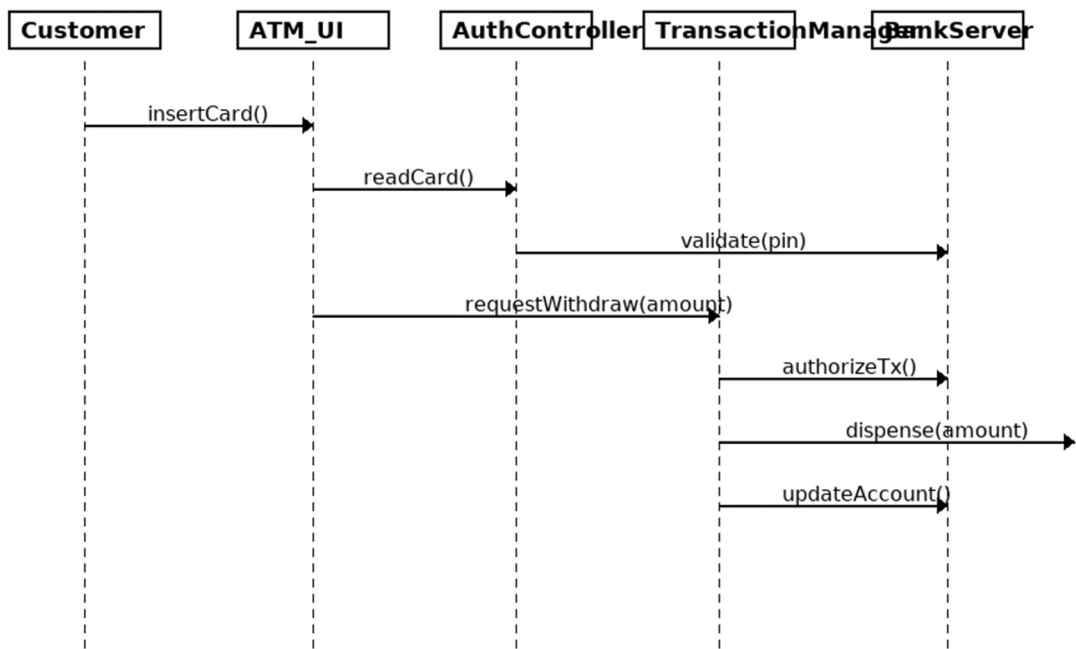
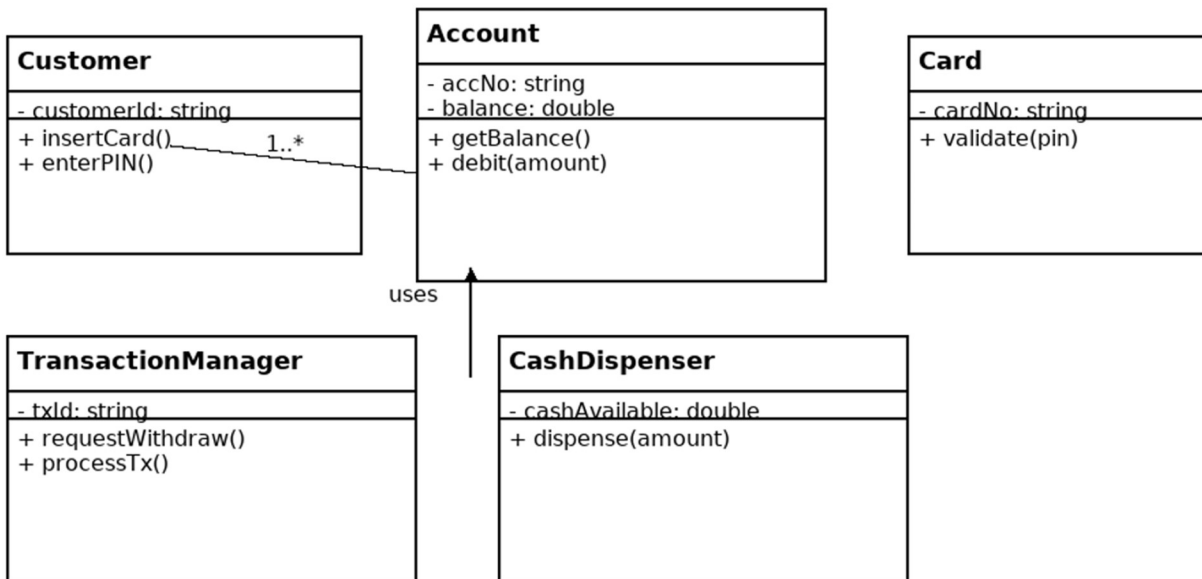
- **Concurrency:** design locks around account updates, use optimistic concurrency or DB transactions.
- **Security:** secure transport (HTTPS), secure storage of PINs (never plaintext), tokenization.
- **Scalability:** separate UI, business, persistence layers; scale stateless services horizontally.
- **Testing:** unit tests for classes, integration tests for use cases, mocks for DAOs and external services.

14) Exam & assignment quick checklist (produce artefacts fast)

1. Write **2–3 use cases** (actor + 1-line main flow).
2. Extract **nouns** → **candidate classes** (6 max).
3. Classify as **boundary/entity/control** (one sentence each).
4. Draw **Class diagram** with 4 classes (attributes + 1 method each) and multiplicities.
5. Draw **Sequence diagram** for the primary use case (4–8 messages).
6. List **2 design axioms** you followed and why.
7. Mention **DAO / persistence choice** in 1–2 lines.
8. If asked, name **pattern(s)** used and where.

Draw the ATM Use Case + Class + Sequence + State diagrams





UNIT II – Object Modeling & Dynamic Modeling

UNIT II:

Object Modeling: Object & classes, Links and Associations, Generalization and Inheritance, Aggregation, Abstract classes, A sample object model, Multiple Inheritance, Meta data, candidate keys, constraints.

Dynamic Modeling: Events and States, Operations and Methods, Nested state Diagrams, Concurrency, Relation of Object and Dynamic Models, advanced dynamic model concepts, a sample dynamic model.


1. Object Modeling

Object modeling represents **real-world entities (objects)** and their **relationships** in the system.

Objects & Classes

- **Object:** A real-world entity with *state* (data/attributes) and *behavior* (methods).
☞ Example: *Student(name, roll_no, marks)*
- **Class:** A blueprint to create objects. Defines attributes & operations.
☞ Example in C++:

cpp

 Copy code

```
class Student {  
    string name;  
    int roll;  
    float marks;  
public:  
    void display();  
};
```

Links and Associations

- **Link:** A physical or conceptual connection between objects.
- **Association:** A relationship type among classes.
☞ Example: *Student ↔ Course* (A student enrolls in a course).

Generalization and Inheritance

- **Generalization:** Abstracting common features into a **superclass**.
- **Inheritance:** A subclass inherits attributes/methods of a superclass.

☞ Example:

- Superclass: Person
 - Subclass: Student, Teacher
-

Aggregation

- A “*whole-part*” relationship where one object contains another, but both can exist independently.

☞ Example: *Library* contains *Books* (if *Library* is deleted, *Books* still exist).

Abstract Classes

- Classes that **cannot be instantiated**.
- Used to define **common behavior** for subclasses.

☞ Example in C++:

```
cpp                                                                    Copy code

class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};
```

Sample Object Model

ATM System:

- Objects: *Customer*, *ATM*, *Account*, *Bank*
- Relationships: *Customer* ↔ *Account*, *ATM* ↔ *Bank*

Multiple Inheritance

- A class inherits from **more than one superclass**.

☞ Example in C++:

cpp

Copy code

```
class Printer { };  
class Scanner { };  
class XeroxMachine : public Printer, public Scanner { };
```

Metadata

- “Data about data” (describes structure, type, constraints).

☞ Example: In DB, *Student(name: string, roll: int)* → metadata = type info.

Candidate Keys & Constraints

- **Candidate Key:** Minimum attribute set uniquely identifying an object.
- **Constraints:** Rules applied to maintain consistency.

☞ Example: Roll number must be unique, age > 0.

2. Dynamic Modeling

Dynamic modeling focuses on **behavior of the system over time**.

Events and States

- **Event:** Something that happens at a point in time (e.g., “Card Inserted”).
 - **State:** Condition of an object at a given time (e.g., “Idle”, “Authenticated”).
-

Operations and Methods

- **Operation:** Functionality available in a class.
- **Method:** Actual implementation of the operation.

Nested State Diagrams

- States may contain **sub-states**.

☞ Example: *ATM* → *Processing* → {*Verifying PIN*, *Checking Balance*}

Concurrency

- Multiple objects or states execute **simultaneously**.

☞ Example: ATM machine allows **printing receipt** while **dispensing cash**.

Relation of Object and Dynamic Models

- Object model → static structure (classes, objects).
 - Dynamic model → behavior (states, events).
 - Together → complete system description.
-

Advanced Dynamic Model Concepts

- **History states** (remember last active sub-state).
 - **Fork/Join states** for parallel processes.
 - **Guard conditions** controlling transitions.
-

Sample Dynamic Model (ATM)

- Events: *Insert card*, *Enter PIN*, *Withdraw cash*
- States: *Idle* → *Card Inserted* → *Validating* → *Transaction* → *Idle*

UNIT III – Functional Modeling & Methodologies

Functional Modeling: Functional Models, Data flow Diagrams, Specifying Operations, Constraints, a sample functional model.

Translating object oriented design into an implementation, OMT Methodologies, examples and case studies to demonstrate methodology, comparison of Methodology, SA/SD, and JSD, Application design and development using OOSD tools.

1. Functional Modeling

Functional modeling describes *what the system should do* (its functionality) without worrying about *how* it will be implemented.

a) Functional Models

- Represent **functions, processes, and transformations** in the system.
 - Focus on **data flow** between processes.
 - Used to understand requirements before moving to design.
-

b) Data Flow Diagrams (DFDs)

- **Purpose:** To show *how data moves* through the system.
- **Elements of DFD:**
 1. **Processes (Circles/Ovals)** → activities/functions performed.
 2. **Data Flows (Arrows)** → movement of data.
 3. **Data Stores (Parallel lines)** → files, databases, storage.
 4. **External Entities (Squares/Rectangles)** → actors outside the system.

◇ Example: ATM System DFD

- Entity: Customer
- Process: Withdraw Money
- Data Store: Bank Database
- Data Flow: Card info, PIN, transaction details.

c) Specifying Operations

- After identifying functions, define **operations**:
 - Input (what data comes in)
 - Process (logic/rules applied)
 - Output (what comes out)

Example: *Withdraw Operation*

- Input → Card number, PIN, amount
 - Process → Validate PIN, check balance, update account
 - Output → Cash, receipt, updated balance
-

d) Constraints

- Rules that must hold true in the system.
 - Types:
 - **Data constraints**: e.g., $\text{balance} \geq 0$.
 - **Operation constraints**: e.g., daily withdrawal limit = ₹25,000.
 - **System constraints**: e.g., ATM must be online.
-

e) Sample Functional Model

A **student information system**:

- Functions → Register student, assign course, generate report.
- DFD shows data flow between student, admin, and database.
- Constraints like *max 5 courses per student per semester*.

2. Translating OOD (Object-Oriented Design) into Implementation

Steps:

1. Map **classes** to code (e.g., Java/C++/Python classes).
 2. Translate **attributes** → **variables**, **methods** → **functions**.
 3. Relationships:
 - Association → references/objects inside other classes.
 - Inheritance → extends/implements in Java.
 - Aggregation/Composition → objects as class members.
-

3. OMT Methodology (Object Modeling Technique)

- Developed by **James Rumbaugh**.
- Uses three main models:
 1. **Object Model** → Classes & objects (static structure).
 2. **Dynamic Model** → State diagrams, event-driven behavior.
 3. **Functional Model** → DFDs, operations.

Phases of OMT:

1. Analysis → Build object, dynamic, functional models.
2. System Design → High-level architecture.
3. Object Design → Add details.
4. Implementation → Code generation.

4. Comparison of Methodologies

Feature	SA/SD (Structured Analysis/Design)	JSD (Jackson Structured Design)	OMT (Object Modeling Technique)
Focus	Processes & data flows	Data structures, sequence of actions	Objects, classes, interactions
Model	DFD, ERD	Structure diagrams, action sequences	Object, functional, dynamic models
Approach	Top-down, functional decomposition	Data-oriented, stepwise refinement	Object-oriented
Suitable for	Traditional systems, batch processing	Business/data processing apps	Complex, real-world systems

5. Application Design & Development using OOSD Tools

- **OOSD (Object-Oriented System Design)** tools help design visually & generate code.
- Tools:
 - **Rational Rose** (UML design tool).
 - **Enterprise Architect**.
 - **StarUML, Visual Paradigm**.
- They allow:
 - Drawing **Use Case, Class, Sequence, State Diagrams**.
 - Auto-generating **skeleton code** in Java/C++/C#.
 - Testing and documentation.

UNIT IV – Programming in C++ , Functions and Overloading

UNIT IV:

Programming in C++ : Introduction OOP Paradigm: Comparison of Programming paradigms, Characteristics of Object-Oriented Programming Languages, Brief History of C++, Structure of a C++ program, Difference between C and C++ - cin, cout, new, delete operators, ANSI/ISO Standard C++, Comments, Working with Variables and const Qualifiers. Enumeration, Arrays and Pointer, Default Parameter Value, Using Reference variables with Functions.

Functions and Overloading: Abstract data types, Class Component, Object & Class, Constructors Default and Copy Constructor, Assignment operator deep and shallow coping, Access modifiers – private, public and protected. Implementing Class Functions within Class declaration or outside the Class declaration. Instantiation of objects, Scope resolution operator, working with Friend Functions, Using Static Class members. Understanding Compile Time Polymorphism function overloading Rules of Operator Overloading (Unary and Binary) as member function/friend function, Implementation of operator overloading of Arithmetic Operators, Overloading Output/Input, Prefix/ Postfix Increment and decrement Operators, Overloading comparison operators, Assignment, subscript and function call Operator , concepts of namespaces.

1. Introduction to OOP Paradigm

a) Comparison of Programming Paradigms

Paradigm	Features	Example
Procedural	Program = sequence of functions; Data is separate	C, Pascal
Object-Oriented	Program = collection of objects; Data + methods together	C++, Java
Functional	Based on mathematical functions; emphasizes immutability	Haskell, Scala
Event-driven	Execution based on events/triggers	JavaScript, GUI apps

b) Characteristics of Object-Oriented Languages

1. **Encapsulation** → data + functions together.
2. **Abstraction** → hide unnecessary details.
3. **Inheritance** → reuse code.
4. **Polymorphism** → many forms (function/operator overloading, virtual functions).

c) Brief History of C++

- Developed by **Bjarne Stroustrup (1980s)** at Bell Labs.
- Extension of C → initially called “C with Classes”.
- Standardized by **ANSI/ISO**.

d) Structure of a C++ Program

```
cpp Copy code

#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!";
    return 0;
}
```

e) Difference between C and C++


Feature	C	C++
Paradigm	Procedural	Object-Oriented
Input/Output	<code>scanf</code> , <code>printf</code>	<code>cin</code> , <code>cout</code>
Memory	<code>malloc</code> , <code>free</code>	<code>new</code> , <code>delete</code>
Data Hiding	Not supported	Supported via classes
Function Overloading	Not available	Available

f) Key Elements

- **cin, cout** → input/output streams.
- **new/delete** → dynamic memory (instead of malloc/free).
- **ANSI/ISO Standard** → ensures portability.
- **Comments** → `//` (single line), `/* */` (multi-line).
- **const Qualifier** → prevents modification.

Example:

cpp


 Copy code

```
const int PI = 3.14;
```

g) Data Types

- **Enumeration:**

cpp

 Copy code

```
enum Days { MON, TUE, WED };  
Days d = MON;
```


Arrays: int arr[5];

Pointers: int *p;

h) Functions

- Default Parameter:


cpp

 Copy code

```
int add(int a, int b=10) { return a+b; }
```

- Reference Parameter:

cpp

 Copy code

```
void swap(int &x, int &y) { int t=x; x=y; y=t; }
```

2. Functions and Overloading

a) Abstract Data Types

- A **class** is an abstract data type → encapsulates data + operations.

```
cpp Copy code  
  
class Stack {  
    int arr[10], top;  
public:  
    Stack() { top = -1; }  
    void push(int x);  
    int pop();  
};
```

b) Class Components

- **Objects** → instances of classes.
- **Constructors** → initialize objects.
- **Destructor** (~) → cleanup.

```
cpp Copy code  
  
class Student {  
    int roll;  
public:  
    Student(int r) { roll = r; } // Constructor  
    ~Student() { cout << "Destroyed"; } // Destructor  
};
```

c) Constructors

- **Default Constructor** → no arguments.
- **Copy Constructor** → copies data.

```
cpp Copy code  
  
Student(const Student &s) {  
    roll = s.roll;  
}
```

d) Assignment Operator

- **Shallow Copy** → copies references only (dangerous with pointers).
 - **Deep Copy** → copies actual values.
-


e) Access Modifiers

- **private**: accessible only inside class.
 - **public**: accessible anywhere.
 - **protected**: accessible in derived classes.
-

f) Scope Resolution Operator (::)

Used to define methods outside class.


```
cpp
void Student::display() { cout << roll; }
```

 Copy code

g) Friend Functions

Allow external function to access private data.


```
cpp
class A {
    int x;
    friend void show(A a);
};
```

 Copy code

h) Static Members

- Belong to class, not objects.

```
cpp
class Test {
    static int count;
};
```


 Copy code

3. Compile-Time Polymorphism

a) Function Overloading

Multiple functions with same name, different parameters.

cpp

 Copy code


```
int add(int a, int b);  
float add(float a, float b);
```

b) Operator Overloading

Modify behavior of operators.

- **Unary Operator:**


cpp

 Copy code

```
class Counter {  
    int value;  
public:  
    void operator++() { value++; } // Prefix ++  
};
```

Binary Operator:


cpp

 Copy code

```
Complex operator+(Complex c);
```

Overloading << and >>:

cpp

 Copy code


```
friend ostream& operator<<(ostream&, Complex&);
```

- **Comparison Operators:** ==, <, etc.
- **Subscript Operator:** obj[i].
- **Function Call Operator:** obj().

4. Namespaces

- Avoids name conflicts.

cpp

 Copy code

```
namespace Math {  
    int add(int a, int b) { return a+b; }  
}  
using namespace Math;
```

UNIT V — Inheritance, Polymorphism, Exceptions, Files, Templates & STL

UNIT V:

Inheritance and Polymorphism: Inheritance, Types of Inheritance, Abstract Classes, Overriding inheritance methods, Constructors and Destructor in derived classes. Multiple Inheritance.

Polymorphism: Polymorphism, Type of Polymorphism – compile time and runtime, Understanding Dynamic polymorphism: Pointer to objects, Virtual Functions (concept of VTABLE) , pure virtual functions, Abstract Class.

Exception Handling and Files: Understanding of working and implementation of Exception Handling, Advanced Input/Output, Manipulating strings, Using istream /ostream member functions, Using Manipulators, Creating Manipulator Functions, Understanding Implementation of Files, Writing and Reading Objects.

Templates: Generic Programming: and mastering STL Understanding Generic Functions with implementation of searching sorting algorithm, Overloading of Function Templates, Standard **Template Library:** Understanding Components of Standard Template Library, Working of Containers, Algorithms, Iterators and Other STL Elements, Implementation of Sequence and Associative containers for different Algorithms using their Iterator.

1. Inheritance (what + types + examples)

Definition: mechanism where a class (derived) reuses/extends features of another class (base).

Why use it: reuse code, model “is-a” relationships, enable polymorphism.

Types

- **Single:** Derived : public Base
- **Multilevel:** C : public B, B : public A
- **Hierarchical:** many derived from one base
- **Multiple:** D : public B, public C
- **Hybrid:** combination (may produce diamond)

Example — single & multilevel

```
cpp                                                                    Copy code

class Person {
public:
    string name;
    Person(string n): name(n) {}
    virtual ~Person() {}
};

class Employee : public Person {
public:
    int empId;
    Employee(string n, int id): Person(n), empId(id) {}
};
```

Multiple inheritance & diamond problem

```
cpp                                                                    Copy code

class A { public: int x; };
class B : virtual public A {}; // virtual inheritance
class C : virtual public A {};
class D : public B, public C {}; // D has one A via virtual
```

- Use **virtual** base to avoid duplicate base subobject and ambiguity.

2. Constructors & Destructors in inheritance

- **Order of construction:** Base constructors run first (top → down), then derived.
- **Destruction:** reverse order — derived destructor then base destructor.

```
cpp                                                                    Copy code

class Base { public: Base(){ cout<<"Base ctor\n"; } virtual ~Base(){ cout<<"Base dtor\n"; } };
class Derived : public Base { public: Derived(){ cout<<"Derived ctor\n"; } ~Derived(){ cout<<"Deri
~Derived(){ cout<<"Derived dtor\n"; } };
```

3. Abstract Classes & Pure Virtual Functions

- **Abstract class:** contains at least one pure virtual function `virtual void f() = 0;` — cannot instantiate.
- **Used to define interfaces.**

```
cpp                                                                    Copy code

class Shape {
public:
    virtual void draw() const = 0; // pure virtual
    virtual ~Shape() = default;
};
class Circle : public Shape {
public:
    void draw() const override { cout << "Draw Circle\n"; }
};
```

4. Polymorphism


a) Compile-time (static)

- **Function overloading, operator overloading, templates.**

b) Run-time (dynamic)

- **Achieved with virtual functions + base pointer/reference to derived object.**

cpp

 Copy code


```
Base* p = new Derived();  
p->virtualMethod(); // calls Derived::virtualMethod() via vtable
```

VTABLE (brief):

- **Compiler generates a vtable (per-class) of function pointers for virtual methods and a vptr (per-object pointer) to that vtable. At runtime the call `p->f()` dispatches via vtable.**

Example with override

cpp

 Copy code

```
class Animal {  
public:  
    virtual void speak() { cout<<"Animal\n"; }  
    virtual ~Animal() = default;  
};  
class Dog : public Animal {  
public:  
    void speak() override { cout<<"Woof\n"; }  
};  
Animal* a = new Dog();  
a->speak(); // "Woof"
```

5. Exception Handling (C++)

Keywords: try, throw, catch.

```
cpp
try {
    if (den == 0) throw runtime_error("Divide by zero");
    cout << num/den;
} catch (const runtime_error &e) {
    cerr << "Error: " << e.what() << '\n';
} catch (...) {
    cerr << "Unknown exception\n";
}
```

Guidelines

- Throw objects (prefer std::exception derived).
- Catch by reference: catch(const std::exception& e).
- Use RAII (Resource Acquisition Is Initialization) to ensure resources free on exceptions (destructors run during stack unwinding).

6. Advanced I/O — Streams, Manipulators, Custom Manipulators

Basic streams

- istream (cin), ostream (cout), fstream (ifstream, ofstream, fstream)

Common member functions

- getline(cin, s), cin.get(), cout.put(), ostream::write() (binary)

Standard manipulators

```
cpp
#include <iomanip>
cout << setw(10) << left << "Name" << setprecision(2) << fixed << 3.14159;
```


Create custom manipulator

```
cpp
ostream& mytab(ostream& os) { return os << '\t'; }
cout << "A" << mytab << "B\n";
```

8. File handling — text & binary, writing/reading objects

Text I/O (portable)

cpp


 Copy code

```
#include <fstream>
ofstream ofs("data.txt");
ofs << "Alice " << 25 << '\n';
ofs.close();

ifstream ifs("data.txt");
string name; int age;
ifs >> name >> age;
```

Binary I/O for POD types (careful with portability)

cpp

 Copy code

```
struct Point { int x,y; };
Point p{10,20};
ofstream bf("p.bin", ios::binary);
bf.write(reinterpret_cast<const char*>(&p), sizeof(p));
bf.close();

Point q;
ifstream br("p.bin", ios::binary);
br.read(reinterpret_cast<char*>(&q), sizeof(q));
```

- For classes with pointers or dynamic memory prefer explicit serialization (text or custom binary protocol).

Writing/Reading objects (example serialization)

```
cpp Copy code

class Person {
    string name;
    int age;
public:
    void save(ofstream &ofs) const {
        size_t n = name.size();
        ofs.write(reinterpret_cast<const char*>(&n), sizeof(n));
        ofs.write(name.data(), n);
        ofs.write(reinterpret_cast<const char*>(&age), sizeof(age));
    }
    void load(ifstream &if_) {
        size_t n;
        if_.read(reinterpret_cast<char*>(&n), sizeof(n));
        name.resize(n);
        if_.read(&name[0], n);
        if_.read(reinterpret_cast<char*>(&age), sizeof(age));
    }
};
```

8. Templates (Generic Programming)

Function template

```
cpp Copy code

template<typename T>
T add(T a, T b) { return a + b; }
int x = add<int>(2,3); // 5
```

Class template

```
cpp Copy code

template<typename T>
class Box {
    T value;
public:
    Box(T v): value(v) {}
    T get() const { return value; }
};

Box<string> b("hi");
```


Template specialization / overloading

- Full or partial specialization to customize behavior for specific types.

9. Standard Template Library (STL) — components & examples

Components


1. Containers — vector, list, deque, set, map, unordered_map, stack, queue, priority_queue.
2. Algorithms — sort, find, binary_search, for_each, accumulate.
3. Iterators — uniform way to traverse containers (begin(), end()).
4. Function objects / lambdas — customization for algorithms.

Sequence vs Associative containers

- Sequence: keep insertion order (vector, list), good for indexed access.
- Associative: key-based lookup (set, map), ordered (RB-tree) or unordered (unordered_map - hash).

Common examples

cpp


 Copy code

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

vector<int> v = {3,1,4,1,5};
sort(v.begin(), v.end());           // 1,1,3,4,5
auto it = find(v.begin(), v.end(), 3);
for(int x : v) cout << x << ' ';
```

Map example


cpp

 Copy code

```
#include <map>
map<string,int> freq;
for(string w : words) ++freq[w];
for(auto &p : freq) cout << p.first << " => " << p.second << '\n';
```

Iterator example

cpp


 Copy code

```
for(auto it = v.begin(); it != v.end(); ++it) cout << *it << ' ';
```

10. Combining templates & STL — generic algorithms

- You can write your own templates that accept iterators to work with any container.

cpp

 Copy code

```
template<typename It>
auto sum_range(It begin, It end) {
    using T = typename std::iterator_traits<It>::value_type;
    T s = T();
    for(; begin!=end; ++begin) s += *begin;
    return s;
}
```

11. Quick pitfalls & best practices

- Use virtual ~Base() when polymorphism expected.
- Avoid slicing: Derived d; Base b = d; — copies only base part.
- Prefer smart pointers (std::unique_ptr, std::shared_ptr) over raw new/delete.
- Prefer std::vector to raw C arrays.
- When serializing, avoid writing raw object memory if it contains pointers; use explicit serialization.
- Catch exceptions by const& and throw objects derived from std::exception for consistent what() messages.

12. Short revision cheatsheet (for exam)

- **Virtual** → runtime dispatch; **override** keyword documents override.
 - **Pure virtual** =0 → abstract class.
 - **VTABLE** → compiler support for virtual calls.
 - **Multiple inheritance** → ambiguity unless resolved (virtual base).
 - **RAII** → destructors free resources even on exception.
 - **fstream** for file IO; `ios::binary` for binary mode.
 - **template<typename T>** → generic function/class.
 - **STL** → vector (contiguous), list (linked), map (ordered), unordered_map (hash).
-

13. Practice problems (try these)

1. Implement Shape base class with pure virtual `area()`. Derive Circle and Rectangle. Show dynamic dispatch using `Shape*`.
2. Serialize a Person class to text file and read it back (handle spaces in name).
3. Write a template function `max_element_custom` that finds maximum in any container using iterators.
4. Demonstrate multiple inheritance ambiguity and resolve it using virtual base class.
5. Use STL to count frequency of words in a file (`unordered_map<string,int>`), then print top 5 frequent words.