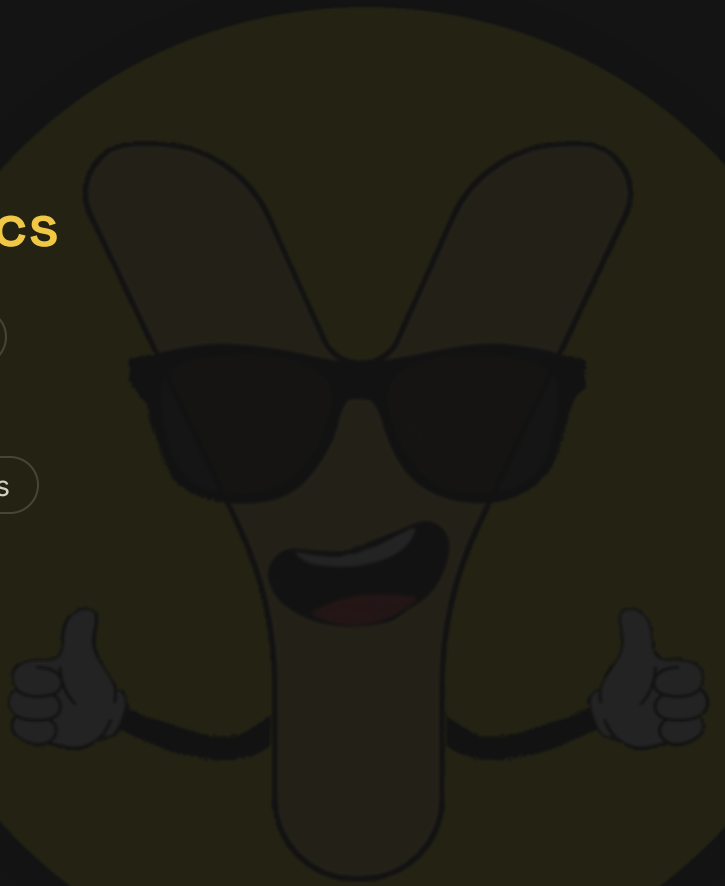
AKTU · 3RD YEAR · SEMESTER 5 · BCS054

Object Oriented System Design with C++

Unit 4 — Object Oriented Programming Style and C++ Basics

- Object oriented programming style: reusability, extensibility, robustness, programming in the large
- Procedural v/s OOP
- Object oriented language features
- Abstraction and Encapsulation
- C++ Basics: Overview
- Program structure
- Namespace
- Identifiers
- Variables
- Constants
- Enum
- Operators
- Typecasting
- Control structures
- C++ Functions: Simple functions
- Call and Return by reference
- Inline functions
- Macro Vs. Inline functions



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

Object-Oriented Programming Style

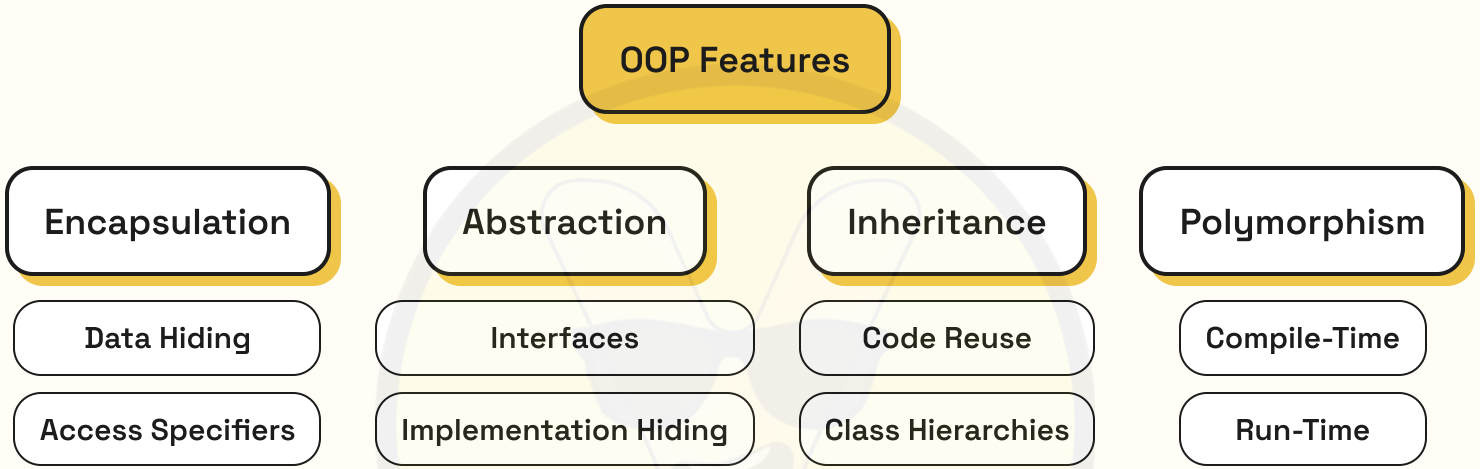
OOP emphasizes key software quality attributes required for enterprise-scale software development.

- Reusability
  - Classes and inheritance allow existing, tested code to be reused without modification.
- Extensibility
  - Systems can be expanded easily by creating new classes or overriding existing behavior.
- Robustness
  - Strong type checking, memory management, and exception handling minimize runtime failures.
- Programming in the Large
  - Modularity allows large software development teams to work on independent modules seamlessly.

Procedural Programming vs Object-Oriented Programming

Feature	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Primary Focus	Functions and algorithms operating on data	Objects combining data and methods
Data Security	Global data is accessible and vulnerable	Data is hidden inside objects (Encapsulation)
Design Approach	Top-down approach	Bottom-up approach
Code Reuse	Limited to function libraries	Achieved via inheritance and polymorphism
Modularity	Structured using functions and files	Structured using classes and objects
Execution Model	Sequential function calls	Message passing between objects

Core Object-Oriented Language Features



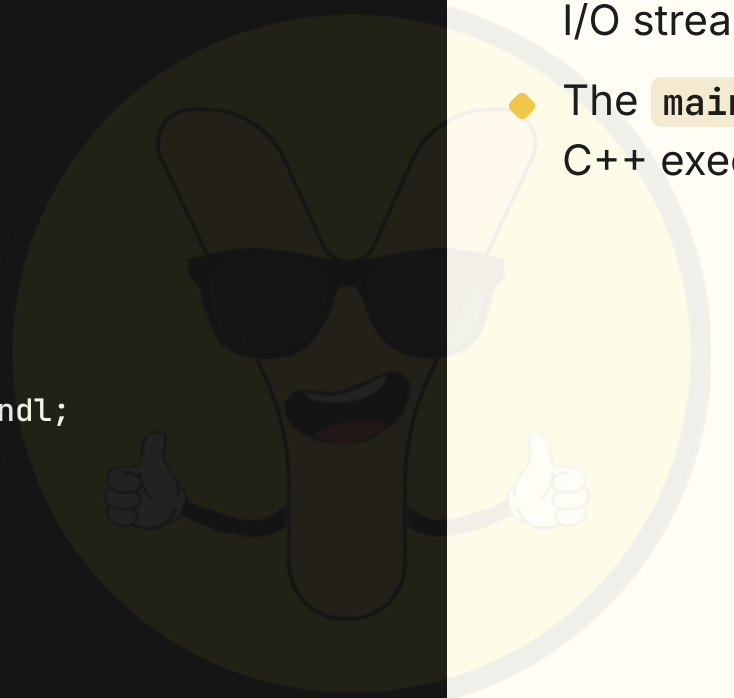
The foundational features defining object-oriented system design and implementation.

Data Abstraction vs. Encapsulation

Property	Data Abstraction	Data Encapsulation
Definition	Hiding complexity and exposing only essential interfaces	Wrapping data and methods together into a single unit
Implementation	Achieved using abstract classes and interfaces	Achieved using private and protected access specifiers
Focus	Focuses on what the object does	Focuses on how data is stored and protected
Design Stage	Concept applied at the design level	Concept applied at the implementation level
Real-world Analogy	Driving a car without knowing internal engine mechanics	Engine components sealed together inside the car bonnet

Basic C++ Program Structure

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

class Demo {
private:
    int value;
public:
    Demo(int v) : value(v) {}
    void show() {
        cout << "Value: " << value << endl;
    }
};

int main() {
    Demo obj(100);
    obj.show();
    return 0;
}
```



- Header files provide declarations for standard I/O streams like `cin` and `cout`.
- The `main()` function is the entry point for C++ execution.

Namespaces in C++

```cpp
#include <iostream>

namespace LibraryA {
    void display() {
        std::cout << "Inside LibraryA" << std::endl;
    }
}

namespace LibraryB {
    void display() {
        std::cout << "Inside LibraryB" << std::endl;
    }
}

int main() {
    LibraryA::display();
    LibraryB::display();
    return 0;
}
```



- Namespaces prevent identifier collisions in large projects using third-party libraries.
- The scope resolution operator `::` specifies which namespace function to execute.

Identifiers, Variables, Constants, and Enums

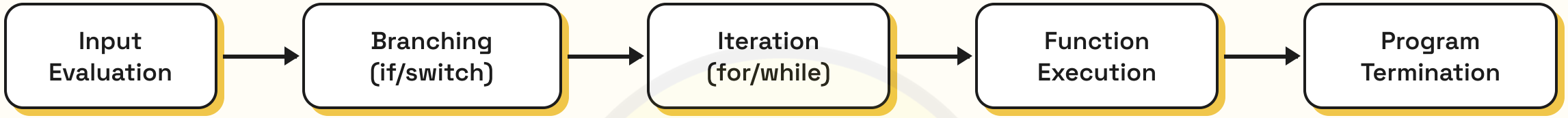
Primitive language constructs used to declare and manipulate data in C++.

- Identifiers
  - User-defined names for variables, functions, and classes; must follow naming rules.
- Variables
  - Named memory locations holding modifiable data of a declared type.
- Constants
  - Immutable values declared using the `const` or `constexpr` keyword.
- Enumerations (enum)
  - User-defined types consisting of named integral constants for improved readability.

C++ Operators and Typecasting

Category	Operators / Methods	Description
Arithmetic & Logical	+, -, *, /, %, &&, ||, !	Standard mathematical and boolean logic operations
Pointer & Memory	&, *, new, delete	Address-of, dereference, dynamic allocation, dynamic release
Scope & Access	::, ., →	Scope resolution, direct member, pointer member access
Implicit Casting	Automatic promotion	Compiler-driven silent conversion (e.g., int to double)
Explicit C-Style	(type) expression	C-style dynamic force conversion without runtime checks
C++ Named Casts	static_cast, dynamic_cast	Type-safe compile-time and runtime polymorphic conversions

Execution Control Structures in C++



Sequential, conditional, and iterative execution paths in a standard C++ program.

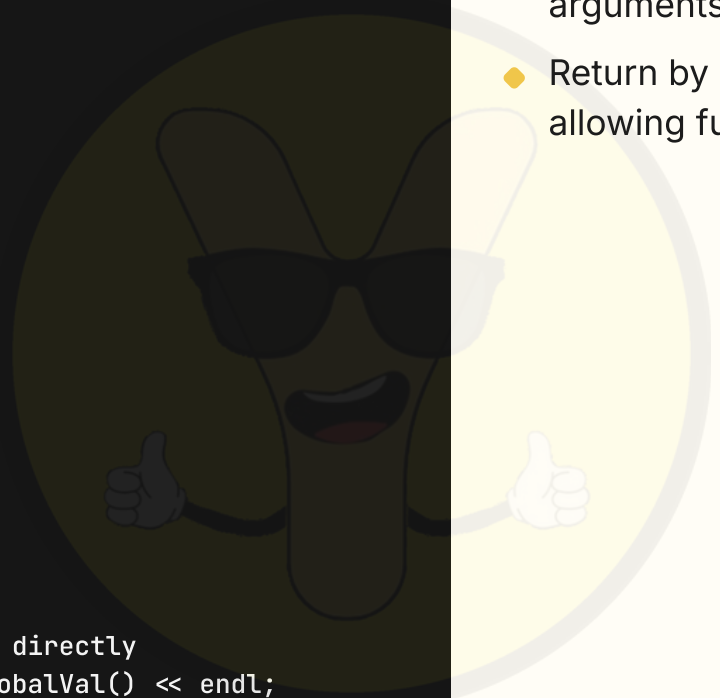
C++ Functions: Fundamentals

Functions break large programs into modular, readable, and reusable units.

- Function Prototype
  - Declares the return type, name, and parameter list before function usage.
- Pass by Value
  - Passes a copy of arguments; changes made inside the function do not affect caller.
- Pass by Pointer
  - Passes memory addresses using pointers, allowing direct modification of actual arguments.
- Stack Overhead
  - Standard functions incur call overhead: pushing parameters, saving registers, and branching.

Call and Return by Reference

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

void swapValues(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

int& getGlobalVal() {
    static int globalVal = 50;
    return globalVal;
}

int main() {
    int x = 10, y = 20;
    swapValues(x, y);
    getGlobalVal() = 500; // Modifies globalVal directly
    cout << "x: " << x << ", global: " << getGlobalVal() << endl;
    return 0;
}
```



- Call by reference passes aliases using `&`, altering caller arguments directly without pointer syntax.
- Return by reference returns an lvalue reference, allowing function calls on the LHS of assignments.

Inline Functions

INLINE FUNCTION

A function for which the compiler expands its code directly at each point of invocation rather than issuing a standard function call.

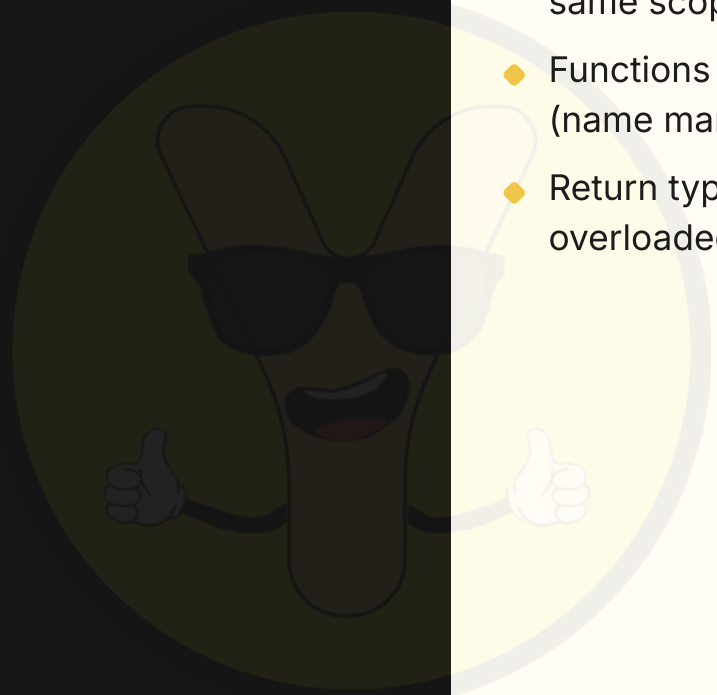
- Declared using the `inline` keyword preceding the function return type.
- Eliminates function call overhead such as parameter pushing and return stack execution.
- Compiler may ignore `inline` requests for large functions, recursive routines, or loops.

Preprocessor Macros vs Inline Functions

Feature	Preprocessor Macro (#define)	Inline Function (inline)
Expansion Time	Expanded by Preprocessor before compilation	Expanded by Compiler during compilation
Type Checking	No type checking (pure string substitution)	Strict type checking performed by compiler
Debugging	Difficult to debug due to macro expansion	Supports standard debugging and breakpoints
Side-effects	Prone to side effects (e.g., `SQR(x++)`)	Safe expression evaluation without unexpected side effects
Scope Rules	Ignores class/namespace scope	Respects class member and namespace access rules
Syntax	Defined with `#define MACRO(x)`	Defined using standard C++ syntax with `inline`

Function Overloading

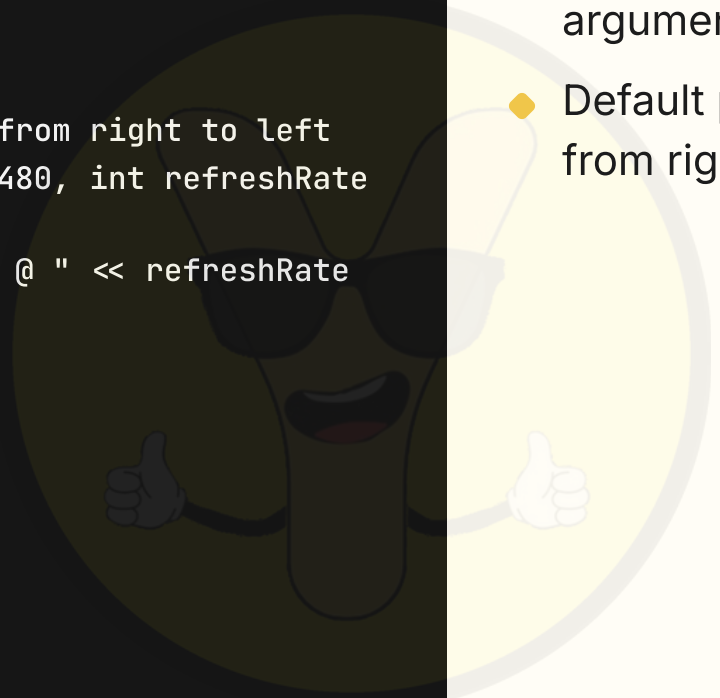
```cpp
#include <iostream>
using namespace std;

int compute(int a, int b) {
    return a + b;
}

double compute(double a, double b) {
    return a * b;
}

int compute(int a, int b, int c) {
    return a + b + c;
}

int main() {
    cout << compute(2, 3) << endl;
    cout << compute(2.5, 4.0) << endl;
    cout << compute(1, 2, 3) << endl;
    return 0;
}
```



- Function overloading allows multiple functions in the same scope to share the same name.
- Functions must differ in parameter count or data types (name mangling resolves calls).
- Return type alone cannot be used to distinguish overloaded functions.

Default Arguments in C++

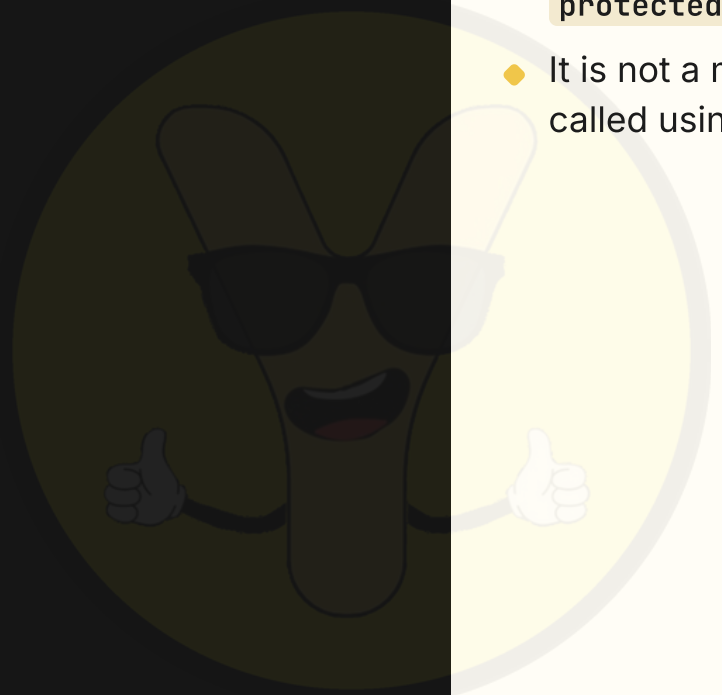
```
#include <iostream>
using namespace std;

// Default parameters must be assigned from right to left
void configure(int width, int height = 480, int refreshRate
= 60) {
    cout << width << "x" << height << " @ " << refreshRate
<< "Hz" << endl;
}

int main() {
    configure(1920);
    configure(1920, 1080);
    configure(1920, 1080, 144);
    return 0;
}
```



- Default arguments provide fallback values if arguments are omitted by the caller.
- Default parameters must be declared trailing from right to left in the signature.

Friend Functions

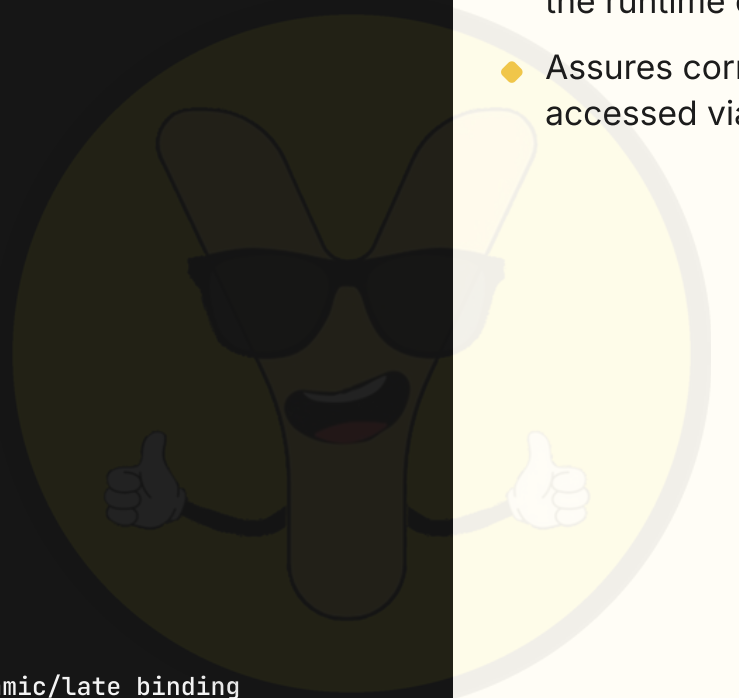
```
#include <iostream>
using namespace std;

class Box {
private:
    double width;
public:
    Box(double w) : width(w) {}
    friend void printWidth(Box b);
};

void printWidth(Box b) {
    // Directly accesses private member 'width'
    cout << "Width: " << b.width << endl;
}

int main() {
    Box box(15.5);
    printWidth(box);
    return 0;
}
```



- A `friend` function is granted access to `private` and `protected` class members.
- It is not a member function of the class and cannot be called using an object handle.

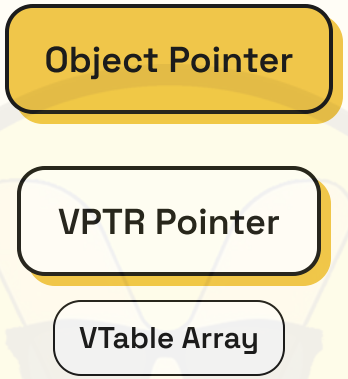
Virtual Functions and Dynamic Polymorphism

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

class Base {
public:
    virtual void render() {
        cout << "Rendering Base" << endl;
    }
};

class Derived : public Base {
public:
    void render() override {
        cout << "Rendering Derived" << endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr→render(); // Calls Derived::render() dynamic/late binding
    delete ptr;
    return 0;
}
```



- `virtual` functions enable dynamic (late) binding based on the runtime object type.
- Assures correct overridden function execution when accessed via a base class pointer.

Virtual Function Internal Architecture (VTable)



Internal resolution mechanism for virtual functions using Virtual Pointer (VPTR) and Virtual Table (VTable).

Exam Corner: Likely AKTU Questions

- **Q1** Differentiate between Object-Oriented Programming and Procedural Programming with neat comparisons.
- **Q2** Explain the concepts of Data Abstraction and Data Encapsulation with real-world and C++ code examples.
- **Q3** What is a Namespace in C++? Illustrate how namespaces prevent identifier naming collisions.
- **Q4** Explain Call by Reference and Return by Reference in C++ with suitable code snippets.
- **Q5** Compare Preprocessor Macros with Inline Functions. Why are Inline Functions preferred in object-oriented code?
- **Q6** What is a Friend Function? Explain its characteristics and code an example demonstrating private member access.
- **Q7** Explain dynamic binding using Virtual Functions. Describe the internal working of VTable and VPTR in C++.

Unit 4 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 5 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

