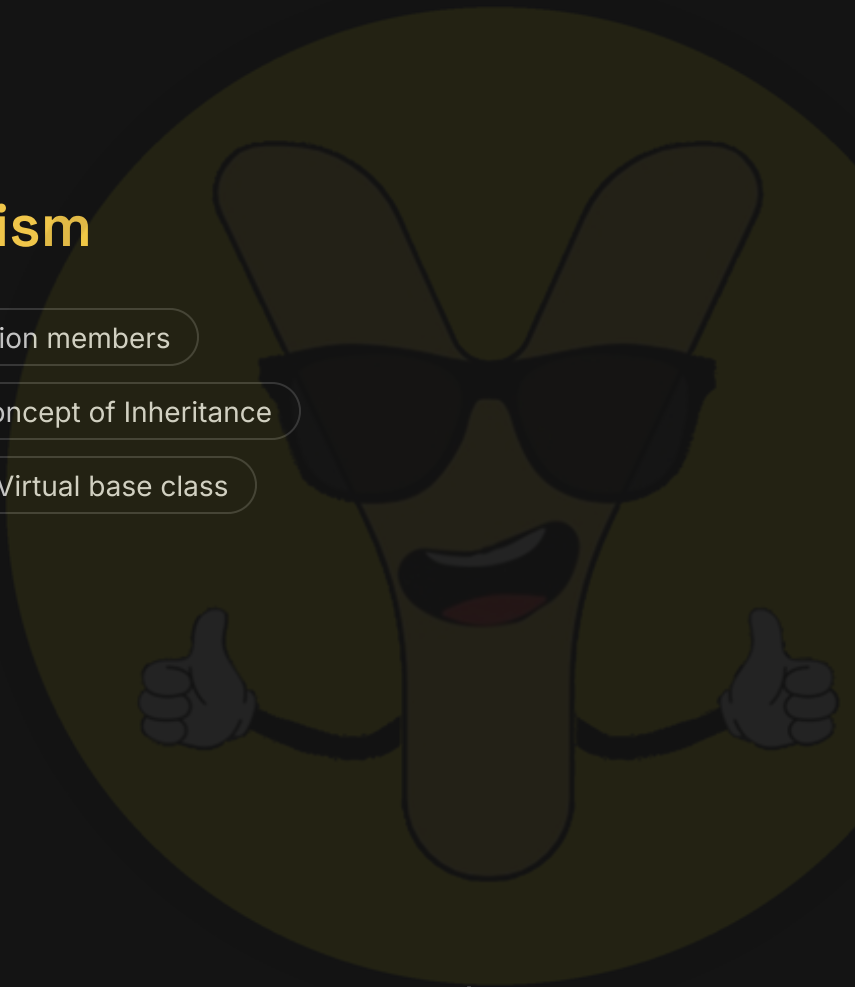
AKTU · 3RD YEAR · SEMESTER 5 · BCS054

Object Oriented System Design with C++

Unit 5 — Objects, Classes, Inheritance and Polymorphism

- Objects and Classes: Basics of object and class in C++
- Private and public members
- Static data and function members
- Constructors and their types
- Destructors
- Operator overloading
- Type conversion
- Inheritance: Concept of Inheritance
- Types of inheritance: single, multiple, multilevel, hierarchical, hybrid
- Protected members
- Overriding
- Virtual base class
- Polymorphism: Pointers in C++
- Pointers and Objects
- this pointer
- Virtual and pure virtual functions



DECODED BY

Chirag Jain

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

Objects and Classes in C++

CLASS & OBJECT

A class is a user-defined data type that encapsulates data members and member functions into a single unit. An object is an instantiated instance of a class allocated in memory at runtime.

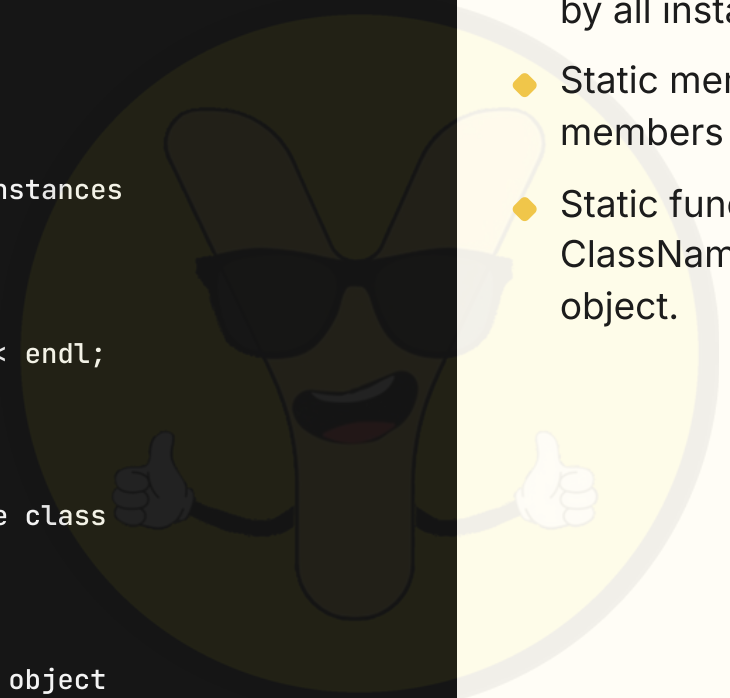
- Classes provide abstraction and encapsulation in C++ programs.
- Objects represent real-world entities containing state (attributes) and behavior (functions).
- Syntax structure: class ClassName { access_specifier: data_members; member_functions(); };

Access Specifiers: Private vs Public vs Protected

Feature	Private	Protected	Public
Accessibility	Inside same class only	Inside class & derived classes	Anywhere in the program
Data Hiding	Highest level of security	Medium level security	No data hiding
Default Status	Default for class members	Must be explicitly declared	Must be explicitly declared
Inheritance	Not inherited in derived class	Inherited via derivation modes	Inherited directly
Direct Access via Object	Not allowed	Not allowed	Allowed using dot operator

Static Data and Static Function Members

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

class Counter {
private:
    static int count; // Shared across all instances
public:
    Counter() { count++; }
    static void displayCount() {
        cout << "Total Objects: " << count << endl;
    }
};

int Counter::count = 0; // Definition outside class

int main() {
    Counter c1, c2;
    Counter::displayCount(); // Call without object
    return 0;
}
```



- Static data members are allocated once and shared by all instances of the class.
- Static member functions can only access static data members and static member functions.
- Static functions are invoked using ClassName::FunctionName() without creating an object.

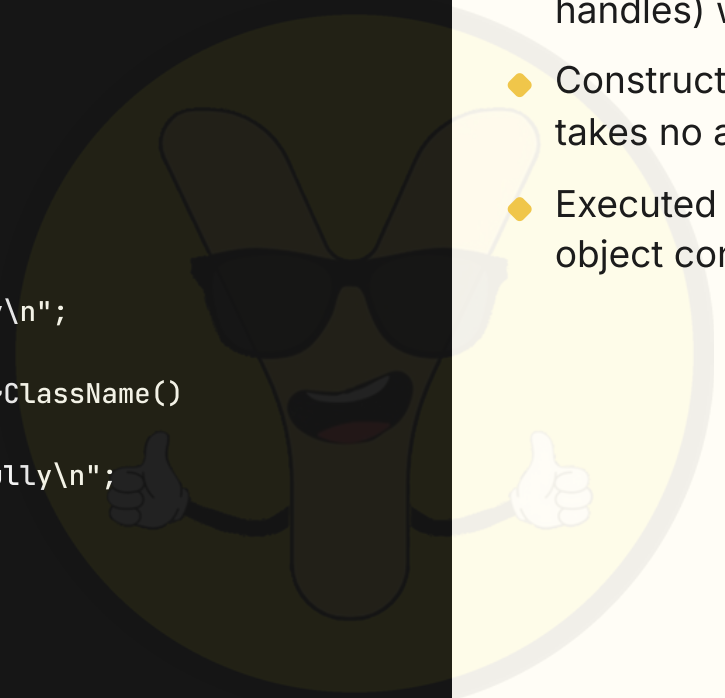
Constructors and Their Classification

Constructors are special member functions executed automatically during object creation to initialize object attributes.

- Default Constructor
  - Takes no arguments for initialization.
  - Automatically generated by the compiler if no explicit constructor is defined.
- Parameterized Constructor
  - Accepts arguments to initialize object attributes with custom values.
  - Supports constructor overloading inside the same class.
- Copy Constructor
  - Initializes an object using another existing object of the same class.
  - Syntax declaration: ClassName(const ClassName &obj);

Destructors in C++

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

class DynamicArray {
    int *ptr;
public:
    DynamicArray(int size) {
        ptr = new int[size];
        cout << "Memory allocated dynamically\n";
    }
    ~DynamicArray() { // Destructor syntax: ~ClassName()
        delete[] ptr;
        cout << "Memory deallocated successfully\n";
    }
};

int main() {
    DynamicArray obj(10);
    return 0; // Destructor called automatically
}
```



- Destructors deallocate resources (memory, file handles) when an object scope ends.
- Constructed with a tilde (~) prefixing the class name, takes no arguments, and returns no value.
- Executed automatically in the exact reverse order of object construction.

Operator Overloading

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

class Complex {
    float real, imag;
public:
    Complex(float r = 0, float i = 0) : real(r), imag(i) {}
    Complex operator + (const Complex &obj) {
        Complex temp;
        temp.real = real + obj.real;
        temp.imag = imag + obj.imag;
        return temp;
    }
    void display() { cout << real << " + " << imag << "i\n"; }
};

int main() {
    Complex c1(2.5, 3.5), c2(1.5, 2.5);
    Complex c3 = c1 + c2; // Triggers operator+ function
    c3.display();
}
```



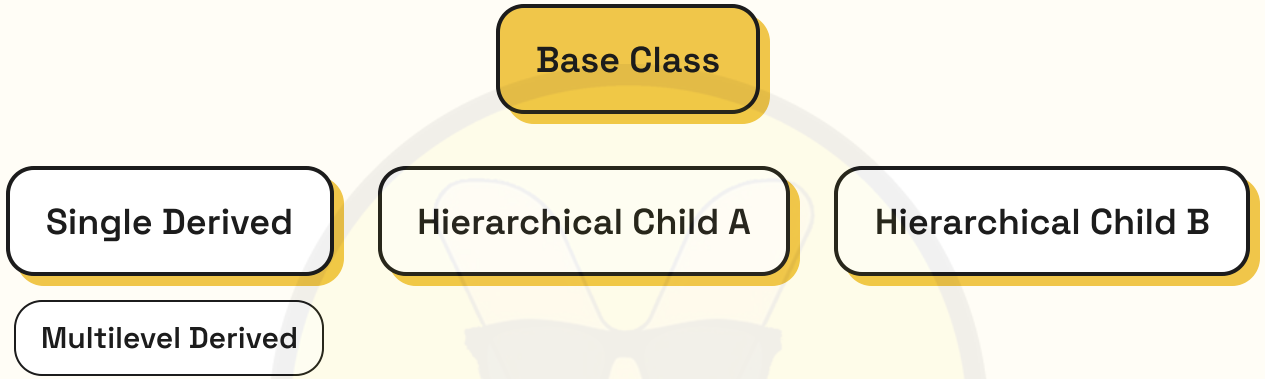
- Redefines C++ operators to work seamlessly with user-defined objects.
- Can be implemented using member functions or non-member friend functions.
- Operators like ::, . , .* , and sizeof cannot be overloaded in C++.

Type Conversion Techniques

Type conversion converts variables or objects from one data type to another during assignments or evaluations.

- Basic Type to Class Type
  - Achieved using a single-argument constructor inside the destination class.
  - Example: Assigning an integer value to a custom Complex class object.
- Class Type to Basic Type
  - Implemented using explicit conversion operator functions inside the class.
  - Syntax: operator basic_type() { return scalar_value; }
- Class Type to Class Type
  - Implemented via conversion constructor in destination class or conversion operator in source class.
  - Enables smooth data transfer between two distinct object classes.

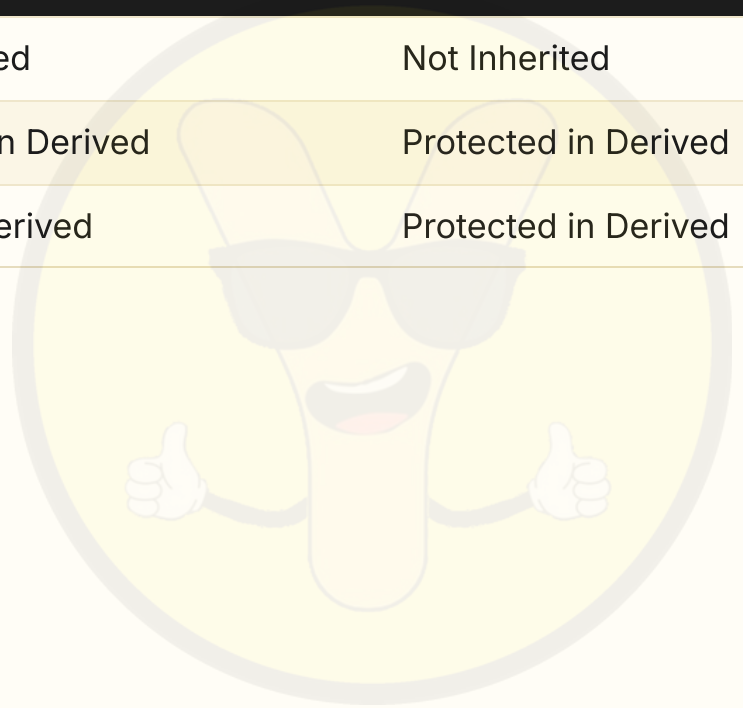
Types of Inheritance Hierarchies



Inheritance models supported in C++: Single, Multilevel, Hierarchical, Multiple, and Hybrid.

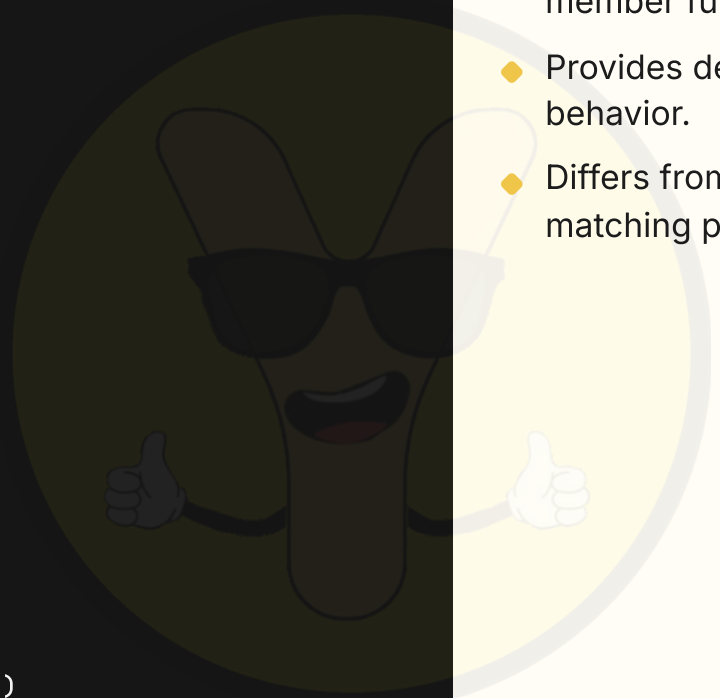
Inheritance Visibility Modes

Base Class Access	Public Derivation	Protected Derivation	Private Derivation
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected in Derived	Protected in Derived	Private in Derived
Public	Public in Derived	Protected in Derived	Private in Derived



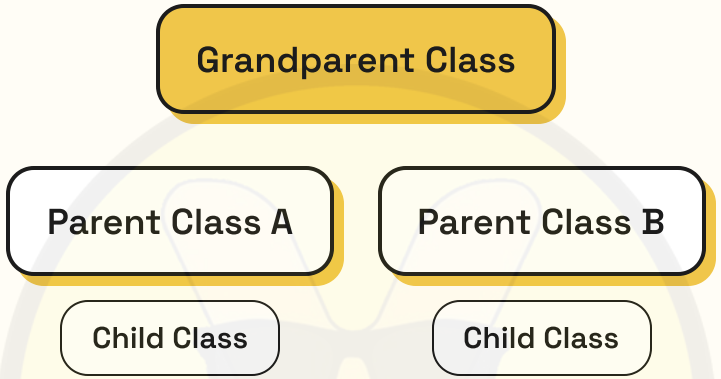
Function Overriding

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

class Base {
public:
    void display() {
        cout << "Base Class Display\n";
    }
};

class Derived : public Base {
public:
    void display() { // Overrides Base::display()
        cout << "Derived Class Display\n";
    }
};

int main() {
    Derived d;
    d.display();        // Calls Derived::display()
    d.Base::display();  // Accesses overridden Base function
    return 0;
}
```



- Occurs when a derived class redefines a base class member function with an identical signature.
- Provides derived classes with specialized operational behavior.
- Differs from overloading: requires inheritance and matching parameter types.

Virtual Base Class (Resolving Diamond Problem)



Declaring virtual base classes prevents duplicate sub-object instances in multipath hybrid inheritance.

Pointers and Objects

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

class Student {
    int rno;
public:
    void setRno(int r) { rno = r; }
    void showRno() { cout << "Roll No: " << rno << endl; }
};

int main() {
    Student s;
    Student *ptr = &s;  // Pointer to object
    ptr->setRno(101);   // Arrow operator access
    ptr->showRno();

    Student *heapObj = new Student(); // Dynamic Object
    heapObj->setRno(102);
    heapObj->showRno();
    delete heapObj;
    return 0;
}
```



- Object pointers store the memory address of instantiated class objects.
- The arrow operator (→) dereferences the pointer to access members directly.
- Facilitates dynamic memory allocation for objects using `new` and `delete`.

The this Pointer

THIS POINTER

An implicit pointer accessible inside non-static member functions pointing directly to the object that invoked the function.

- Disambiguates instance attributes from local function arguments with identical names.
- Enables method chaining by returning the dereferenced object (return *this).
- Unavailable inside static functions because static functions operate without object instances.

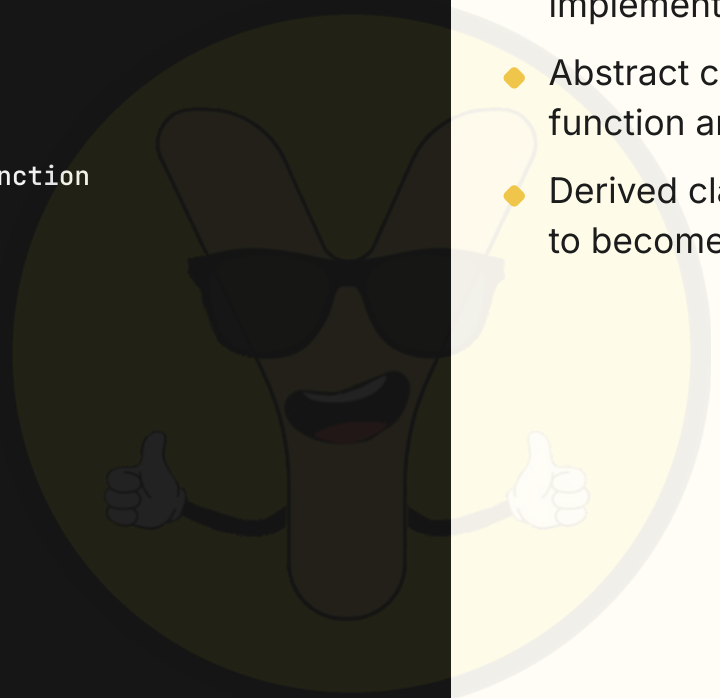
Virtual Functions and Dynamic Binding

Virtual functions enable runtime polymorphism by resolving function calls during program execution rather than compilation.

- Virtual Keyword
  - Declared in base class using the `virtual` keyword.
  - Ensures overridden derived version executes when invoked via a base pointer.
- Dynamic / Late Binding
  - Compiler binds function calls at runtime based on the actual object target.
  - Replaces compile-time static binding.
- Execution Rules
  - Must be member functions of a class.
  - Constructors cannot be virtual, but destructors can and often should be virtual.

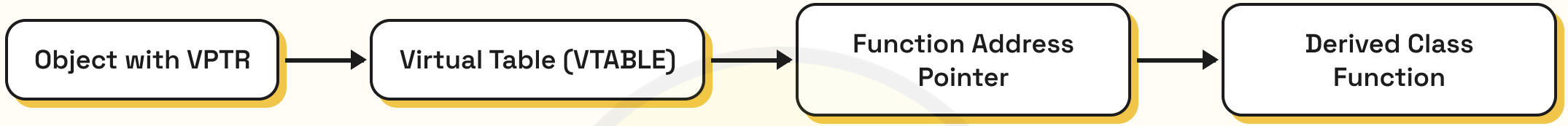
Pure Virtual Functions and Abstract Classes

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

class Shape { // Abstract Class
public:
    virtual void draw() = 0; // Pure Virtual Function
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle\n";
    }
};

int main() {
    Shape *ptr = new Circle();
    ptr→draw(); // Dynamic dispatch
    delete ptr;
    return 0;
}
```



- Pure virtual functions have `= 0` declarations and no implementation in base classes.
- Abstract classes contain at least one pure virtual function and cannot be instantiated.
- Derived classes must override all pure virtual functions to become concrete classes.

Polymorphism Implementation: VPTR & VTABLE



Compilers build VTABLEs containing virtual function addresses and inject a hidden VPTR into objects.

Compile-Time vs Run-Time Polymorphism

Parameter	Compile-Time Polymorphism	Run-Time Polymorphism
Mechanism	Function & Operator Overloading	Virtual Functions & Overriding
Binding Type	Early / Static Binding	Late / Dynamic Binding
Execution Speed	Faster execution	Slightly slower due to indirection
Flexibility	Less flexible during execution	Highly dynamic and flexible
Inheritance	Not mandatory	Strictly mandatory requirement
Invocation	Invoked via standard objects	Invoked via base class pointers/references

Exam Corner: Likely AKTU Questions

- **Q1** Differentiate between class and object. Write a program demonstrating static data members and static functions in C++.
- **Q2** What are constructors and destructors? Explain copy constructors with a code snippet.
- **Q3** Explain operator overloading. Write a program to overload binary '+' operator for adding two complex numbers.
- **Q4** Explain type conversion techniques between basic data types and user-defined class objects with examples.
- **Q5** Describe all inheritance types in C++ with clear structural diagrams and access specifier rules.
- **Q6** What is the diamond problem in hybrid inheritance? How does a virtual base class resolve it?
- **Q7** Differentiate between compile-time and runtime polymorphism with clear examples.
- **Q8** Explain virtual functions, pure virtual functions, and abstract classes. How do VPTR and VTABLE work internaly?

Unit 5 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 1 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

