Ultimate OOP C++ Revision Guide

Complete Syllabus Coverage for MSBTE Students

Read Time: 2 Hours | Perfect for Last-Day Revision

📚 UNIT I - PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

1. Core Definitions (Exam-Ready)

Object-Oriented Programming (OOP): A programming paradigm that uses objects and classes to design applications and computer programs, promoting code reusability and modularity.

Procedural Programming (POP): A programming approach that uses functions or procedures to perform operations on data, following a top-down approach.

Class: A blueprint or template that defines the properties and behaviors (data members and member functions) that objects of that type will have.

Object: An instance of a class that occupies memory and can perform operations defined in the class.

Encapsulation: The mechanism of binding data (variables) and functions that operate on the data together within a single unit (class), restricting direct access to internal implementation.

Inheritance: The process by which one class (derived/child class) acquires properties and methods of another class (base/parent class).

Polymorphism: The ability of objects of different types to respond to the same interface in different ways, allowing the same function name to work differently based on context.

Abstraction: The process of hiding complex implementation details while showing only essential features to the user.

2. POP vs OOP Comparison

Aspect	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Approach	Top-down approach	Bottom-up approach
Focus	Functions and procedures	Objects and classes
Data Security	Less secure (global data)	More secure (encapsulation)
Code Reusability	Limited	High (through inheritance)
Problem Solving	Divides problem into functions	Divides problem into objects
Example Languages	C, Pascal, COBOL	C++, Java, Python
Data and Functions	Separate	Combined in classes
Maintenance	Difficult for large programs	Easier to maintain

3. Features of OOP

Encapsulation:

- Wraps data and methods together in a class
- Example: A `BankAccount` class hides balance details and provides methods like `deposit()` and `withdraw()`

Inheritance:

- Enables code reuse by inheriting properties from parent class
- Example: `Car` class inherits from `Vehicle` class

Polymorphism:

- Same function name behaves differently for different objects
- Example: `draw()` function works differently for `Circle` and `Rectangle` objects

Abstraction:

- Hides unnecessary details from user
- Example: Using ATM without knowing internal processing

4. C++ Special Operators

Scope Resolution Operator (::):

- Used to access global variables, static members, or members of a specific class
- Syntax: `className::memberName` or `::globalVariable`

Memory Management Operators:

- `new`: Allocates memory dynamically at runtime

- `delete`: Deallocates memory allocated by `new`
- Example: `int *ptr = new int(10);` then `delete ptr;`

Manipulators:

- Special functions used to format input/output
- Common ones: `endl`, `setw()`, `setprecision()`, `fixed`

5. Data Types and Variables

Type Casting: Converting one data type to another

- **Implicit:** Automatic conversion (e.g., int to float)
- **Explicit:** Manual conversion using cast operators

Reference Variable: An alias or alternative name for an existing variable

- Syntax: `int &ref = variable;`
- Must be initialized during declaration

Dynamic Initialization: Initializing variables at runtime rather than compile time

6. Structure of C++ Program

```cpp
#include <iostream>       // Header files
using namespace std;      // Namespace declaration

class ClassName {         // Class definition
private:                  // Access specifier
    int data;             // Data members
public:
    void function();      // Member functions
};

int main() {              // Main function
    // Program execution starts here
    return 0;             // Return statement
}
```

7. Class and Object Details

Access Specifiers:

- **Private:** Accessible only within the same class
- **Public:** Accessible from anywhere in the program
- **Protected:** Accessible within the same class and derived classes

Defining Member Functions:

- **Inside Class:** Function definition written inside class body
- **Outside Class:** Function definition written outside using scope resolution operator

Memory Allocation: Each object gets separate memory for data members, but shares member functions

---

📚 UNIT II - FUNCTIONS AND CONSTRUCTORS

8. Function Types

Inline Function: A function whose code is expanded at the point of call instead of calling the function, reducing function call overhead.

- Use `inline` keyword
- Best for small, frequently called functions

Static Data Member: A data member that is shared by all objects of the class, having only one copy regardless of number of objects created.

Static Member Function: A function that belongs to the class rather than any specific object and can only access static data members.

Friend Function: A non-member function that has access to private and protected members of a class.

9. Constructor and Destructor

Constructor: A special member function that is automatically called when an object is created to initialize the object.

- Same name as class
- No return type
- Cannot be virtual

Destructor: A special member function that is automatically called when an object goes out of scope to clean up resources.

- Same name as class with `~` prefix

- No return type or parameters
- Only one destructor per class

10. Types of Constructors

Default Constructor: Takes no parameters and provides default initialization

```cpp
Student() { name = "Unknown"; rollNo = 0; }
```

Parameterized Constructor: Takes parameters to initialize object with specific values

```cpp
Student(string n, int r) { name = n; rollNo = r; }
```

Copy Constructor: Creates a new object as a copy of an existing object

```cpp
Student(const Student &s) { name = s.name; rollNo = s.rollNo; }
```

11. Constructor vs Destructor Comparison

Aspect	Constructor	Destructor
Purpose	Initialize objects	Clean up resources
When Called	Object creation	Object destruction
Parameters	Can have parameters	No parameters
Overloading	Can be overloaded	Cannot be overloaded
Return Type	No return type	No return type
Symbol	Same as class name	~ + class name

12. Array of Objects

Creating multiple objects of the same class and storing them in an array:

```cpp
Student students[50];  // Array of 50 Student objects
```

📚 UNIT III - INHERITANCE

13. Inheritance Fundamentals

Base Class (Parent Class): The class whose properties are inherited by another class.

Derived Class (Child Class): The class that inherits properties from another class.

Visibility Modes: Control access to inherited members

- **Public Inheritance:** Public members remain public, protected remain protected
- **Private Inheritance:** All members become private
- **Protected Inheritance:** Public and protected members become protected

14. Types of Inheritance

Single Inheritance: One derived class inherits from one base class

- Example: `Car` inherits from `Vehicle`

Multilevel Inheritance: A derived class becomes base class for another class

- Example: `Vehicle` → `Car` → `SportsCar`

Multiple Inheritance: One derived class inherits from multiple base classes

- Example: `Child` inherits from both `Father` and `Mother`

Hierarchical Inheritance: Multiple derived classes inherit from single base class

- Example: `Car`, `Bike`, `Truck` all inherit from `Vehicle`

Hybrid Inheritance: Combination of two or more types of inheritance

---

💻 ESSENTIAL CODE EXAMPLES

Program 1: Memory Management with new and delete

```cpp
```

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

int main() {
    // Dynamic memory allocation
    int *ptr = new int(25);
    cout << "Value: " << *ptr << endl;
    cout << "Address: " << ptr << endl;

    // Memory deallocation
    delete ptr;
    ptr = nullptr; // Good practice

    // Array allocation
    int *arr = new int[5];
    for(int i = 0; i < 5; i++) {
        arr[i] = i + 1;
        cout << arr[i] << " ";
    }
    delete[] arr; // Array deallocation

    return 0;
}
```

Program 2: Student Class with Member Function Inside Class

cpp

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

class Student {
private:
    string name;
    int rollNo;

public:
    // Member function defined inside class
    void getData() {
        cout << "Enter name: ";
        cin >> name;
        cout << "Enter roll number: ";
        cin >> rollNo;
    }

    void displayData() {
        cout << "Name: " << name << endl;
        cout << "Roll No: " << rollNo << endl;
    }
};

int main() {
    Student s1;
    s1.getData();
    s1.displayData();
    return 0;
}
```

Program 3: Difference between struct and class

```cpp
```

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

// In struct, members are public by default
struct StudentStruct {
    int rollNo;  // Public by default
    void display() {
        cout << "Struct Roll No: " << rollNo << endl;
    }
};

// In class, members are private by default
class StudentClass {
    int rollNo;  // Private by default
public:
    void setRollNo(int r) { rollNo = r; }
    void display() {
        cout << "Class Roll No: " << rollNo << endl;
    }
};

int main() {
    StudentStruct s1;
    s1.rollNo = 101;  // Direct access possible
    s1.display();

    StudentClass s2;
    // s2.rollNo = 102;  // Error: private member
    s2.setRollNo(102);   // Must use public function
    s2.display();

    return 0;
}
```

Program 4: Complex Number Addition

cpp

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

class Complex {
private:
    float real, imag;

public:
    void getData() {
        cout << "Enter real part: ";
        cin >> real;
        cout << "Enter imaginary part: ";
        cin >> imag;
    }

    Complex add(Complex c2) {
        Complex temp;
        temp.real = real + c2.real;
        temp.imag = imag + c2.imag;
        return temp;
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1, c2, result;

    cout << "Enter first complex number:" << endl;
    c1.getData();

    cout << "Enter second complex number:" << endl;
    c2.getData();

    result = c1.add(c2);

    cout << "Sum: ";
    result.display();

    return 0;
}
```

Program 5: Friend Function for Average Calculation

cpp

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

class Test2; // Forward declaration

class Test1 {
private:
    int marks1;

public:
    void getData() {
        cout << "Enter marks for Test1: ";
        cin >> marks1;
    }

    friend float calculateAverage(Test1 t1, Test2 t2);
};

class Test2 {
private:
    int marks2;

public:
    void getData() {
        cout << "Enter marks for Test2: ";
        cin >> marks2;
    }

    friend float calculateAverage(Test1 t1, Test2 t2);
};

float calculateAverage(Test1 t1, Test2 t2) {
    return (t1.marks1 + t2.marks2) / 2.0;
}

int main() {
    Test1 t1;
    Test2 t2;

    t1.getData();
    t2.getData();

    float avg = calculateAverage(t1, t2);
    cout << "Average marks: " << avg << endl;
```

```
    return 0;
}
```

Program 6: Calculator using Friend Function

cpp

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

class Calculation {
private:
    float num1, num2;

public:
    void getData() {
        cout << "Enter two numbers: ";
        cin >> num1 >> num2;
    }

    friend void performOperations(Calculation c);
};

void performOperations(Calculation c) {
    cout << "Addition: " << c.num1 + c.num2 << endl;
    cout << "Subtraction: " << c.num1 - c.num2 << endl;
    cout << "Multiplication: " << c.num1 * c.num2 << endl;
    if(c.num2 != 0)
        cout << "Division: " << c.num1 / c.num2 << endl;
    else
        cout << "Division: Cannot divide by zero" << endl;
}

int main() {
    Calculation calc;
    calc.getData();
    performOperations(calc);
    return 0;
}
```

Program 7: Simple Interest with Static Member

cpp

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

class SimpleInterest {
private:
    float principal, time;
    static float rate_of_interest; // Static member

public:
    void getData() {
        cout << "Enter principal amount: ";
        cin >> principal;
        cout << "Enter time (years): ";
        cin >> time;
    }

    float calculateSI() {
        return (principal * rate_of_interest * time) / 100;
    }

    void display() {
        cout << "Principal: " << principal << endl;
        cout << "Time: " << time << " years" << endl;
        cout << "Rate: " << rate_of_interest << "%" << endl;
        cout << "Simple Interest: " << calculateSI() << endl;
    }
};

// Static member initialization
float SimpleInterest::rate_of_interest = 7.5;

int main() {
    SimpleInterest si1, si2;

    cout << "For first person:" << endl;
    si1.getData();
    si1.display();

    cout << "\nFor second person:" << endl;
    si2.getData();
    si2.display();

    return 0;
}
```

Program 8: Constructor with Default Course

cpp

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

class Student {
private:
    string name;
    int rollNo;
    string course;

public:
    // Constructor with default course
    Student() {
        course = "Computer Engineering";
        cout << "Enter student name: ";
        cin >> name;
        cout << "Enter roll number: ";
        cin >> rollNo;
    }

    void display() {
        cout << "\n--- Student Details ---" << endl;
        cout << "Name: " << name << endl;
        cout << "Roll No: " << rollNo << endl;
        cout << "Course: " << course << endl;
    }
};

int main() {
    cout << "Creating student object..." << endl;
    Student s1;
    s1.display();
    return 0;
}
```

Program 9: Parameterized Constructor

cpp

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

class Student {
private:
    string name;
    float percentage;

public:
    // Parameterized constructor
    Student(string n, float p) {
        name = n;
        percentage = p;
        cout << "Student object created successfully!" << endl;
    }

    void display() {
        cout << "\n--- Student Information ---" << endl;
        cout << "Name: " << name << endl;
        cout << "Percentage: " << percentage << "%" << endl;

        if(percentage >= 75)
            cout << "Grade: Distinction" << endl;
        else if(percentage >= 60)
            cout << "Grade: First Class" << endl;
        else if(percentage >= 50)
            cout << "Grade: Second Class" << endl;
        else
            cout << "Grade: Pass" << endl;
    }
};

int main() {
    string studentName;
    float marks;

    cout << "Enter student name: ";
    cin >> studentName;
    cout << "Enter percentage: ";
    cin >> marks;

    Student s1(studentName, marks);
    s1.display();
```

```
    return 0;
}
```

Program 10: Time Class with Constructor

cpp

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

class Time {
private:
    int hrs, min, sec;

public:
    // Default constructor
    Time() {
        hrs = 0;
        min = 0;
        sec = 0;
    }

    // Parameterized constructor
    Time(int h, int m, int s) {
        hrs = h;
        min = m;
        sec = s;
        normalizeTime(); // Ensure valid time format
    }

    void normalizeTime() {
        if(sec >= 60) {
            min += sec / 60;
            sec = sec % 60;
        }
        if(min >= 60) {
            hrs += min / 60;
            min = min % 60;
        }
        if(hrs >= 24) {
            hrs = hrs % 24;
        }
    }

    void display() {
        cout << "Time: ";
        if(hrs < 10) cout << "0";
        cout << hrs << ":";
        if(min < 10) cout << "0";
        cout << min << ":";
        if(sec < 10) cout << "0";
```

```
        cout << sec << endl;
    }

    Time addTime(Time t2) {
        Time result;
        result.sec = sec + t2.sec;
        result.min = min + t2.min;
        result.hrs = hrs + t2.hrs;
        result.normalizeTime();
        return result;
    }
};

int main() {
    Time t1; // Default constructor
    cout << "Default time: ";
    t1.display();

    Time t2(14, 30, 45); // Parameterized constructor
    cout << "Time 2: ";
    t2.display();

    Time t3(10, 35, 20);
    cout << "Time 3: ";
    t3.display();

    Time sum = t2.addTime(t3);
    cout << "Sum of Time 2 and Time 3: ";
    sum.display();

    return 0;
}
```

Program 11: Inheritance - Employee to Salary

```
cpp
```

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

// Base class
class Employee {
protected:
    string name;
    int empId;

public:
    void getEmployeeData() {
        cout << "Enter employee name: ";
        cin >> name;
        cout << "Enter employee ID: ";
        cin >> empId;
    }

    void displayEmployeeData() {
        cout << "Employee Name: " << name << endl;
        cout << "Employee ID: " << empId << endl;
    }
};

// Derived class
class Salary : public Employee {
private:
    float basicSalary;
    float hra, da, pf;
    float netSalary;

public:
    void getSalaryData() {
        getEmployeeData(); // Call base class function
        cout << "Enter basic salary: ";
        cin >> basicSalary;
    }

    void calculateSalary() {
        hra = basicSalary * 0.20;  // 20% HRA
        da = basicSalary * 0.10;   // 10% DA
        pf = basicSalary * 0.08;   // 8% PF (deduction)

        netSalary = basicSalary + hra + da - pf;
```

```
    }

    void displaySalary() {
        displayEmployeeData(); // Call base class function
        cout << "\n--- Salary Details ---" << endl;
        cout << "Basic Salary: Rs. " << basicSalary << endl;
        cout << "HRA (20%): Rs. " << hra << endl;
        cout << "DA (10%): Rs. " << da << endl;
        cout << "PF Deduction (8%): Rs. " << pf << endl;
        cout << "Net Salary: Rs. " << netSalary << endl;
    }
};

int main() {
    Salary emp;

    cout << "=== Employee Salary Calculation ===" << endl;
    emp.getSalaryData();
    emp.calculateSalary();

    cout << "\n=== Employee Details ===" << endl;
    emp.displaySalary();

    return 0;
}
```

🎯 EXAM STRATEGY & QUICK TIPS

Time Management (2.5 Hours = 150 minutes)

- Q1 (10M): 15 minutes - Quick definitions and short answers
- Q2-Q4 (36M): 90 minutes - 30 minutes each for detailed explanations
- Q5-Q6 (24M): 45 minutes - 22.5 minutes each for complete programs

Writing Best Practices

- **Always start with definitions** - Never lose definition marks
- **Use proper headings** - Make answers organized and readable
- **Include complete programs** - Don't write code snippets for 6M questions
- **Add comments in code** - Shows understanding
- **Write expected output** - Demonstrates program execution

Common Mistakes to Avoid

- ❌ Forgetting semicolon after class definition
- ❌ Not initializing static members outside class
- ❌ Mixing up constructor and destructor syntax
- ❌ Writing incomplete programs for programming questions
- ❌ Not explaining program logic

High-Priority Topics for Revision

1. **Constructors and Destructors** (Types, syntax, usage)
2. **Friend Functions** (Concept and implementation)
3. **Static Members** (Static data and function members)
4. **Inheritance** (Types and implementation)
5. **Access Specifiers** (Private, public, protected)
6. **Memory Management** (new and delete operators)

Golden Rules for Maximum Marks

- Write clean, commented code
- Always include #include statements and main function
- Explain your logic after writing programs
- Use meaningful variable names
- Show sample input/output for programs

Remember: Practice writing complete answers within time limits. This guide covers all syllabus topics with exam-friendly definitions and practical examples. Focus on understanding concepts rather than memorizing, and you'll excel in your OOP C++ examination!

Best of luck for your exams! 🌟