VG

MCA-102

OBJECT ORIENTED PROGRAMMING WITH C++

PREFACE

Dear Reader,

It is a privilege to present this comprehensive collection of **RTU MCA Semester Examination Notes**, designed to cover the complete syllabus with clarity and academic accuracy. Every concept, definition, explanation, and example has been organized to support thorough understanding and effective exam preparation.

The purpose of these notes is to simplify complex topics and provide a reliable study resource that can assist in both detailed learning and quick revision. Continuous effort has been made to ensure correctness and relevance; however, learning grows when readers engage, question, and explore further.

May these notes serve as a strong academic foundation and contribute meaningfully to your preparation and future growth.

Warm regards,
Virendra Goura
Author
www.virendragoura.com

DISCLAIMER

This e-book has been created with utmost care, sincere effort, and extensive proofreading. However, despite all attempts to avoid mistakes, there may still be some errors, omissions, or inaccuracies that remain unnoticed.
This e-book is issued with the understanding that neither the author nor the publisher shall be held responsible for any loss, damage, or misunderstanding arising from the use of the information contained within.
All content provided is for educational and informational purposes only.

MCA-102 Object Oriented Programming with C++

Unit-1

OOP Paradigm:

Characteristics of OOP, Comparison between functional programming and OOP approach, characteristics of object oriented language - objects, classes, inheritance, reusability, user defined data types, polymorphism, overloading.

Unit-2

Introduction to C++:

Identifier and keywords, constants, C++ operators, type conversion, Variable declaration, statements, expressions, input and output, conditional expression loop statements, break control statements, Classes, member functions, objects, arrays of class objects, pointers and classes, nested classes, constructors, destructors Inline member functions, static class member, friend functions, and dynamic memory allocation.

Unit-3

Polymorphism and Inheritance:

Function overloading, operator overloading, polymorphism, early binding, polymorphism with pointers, virtual functions, late binding, pure virtual functions.

Single inheritance, types of inheritance, types of base classes, types of derivations, multiple inheritances, container classes, member access control.

Unit-4

Exceptions and Templates:

Exception Syntax, Multiple Exceptions, Function Templates, Function Templates with multiple argument templates.

Unit-5

File Handling in C++:

C++ Streams, Console Stream Classes, Formatted And Unformatted Console I/O Operations, manipulators, File Streams, Classes File Modes, File Pointers and Manipulations File I/O.

Unit-1

OOP Paradigm:

Characteristics of OOP, Comparison between functional programming and OOP approach, characteristics of object oriented language - objects, classes, inheritance, reusability, user defined data types, polymorphism, overloading.

The Object-Oriented Programming (OOP) Paradigm is a programming approach where software is built using objects and classes instead of only functions. It focuses on organizing data and behavior together inside objects.

Characteristics of OOP

Object-Oriented Programming (OOP) provides several important characteristics that help in building structured and efficient programs. The major characteristics include **Objects, Classes, Encapsulation, Abstraction, Inheritance, and Polymorphism**.

- **Objects** represent real-world entities and contain both data and behavior.
- **Classes** act as templates or blueprints used to create objects.
- **Encapsulation** ensures that data and methods are bundled together within an object, providing data protection and controlled access.
- **Abstraction** hides unnecessary implementation details and shows only essential features to the user.
- **Inheritance** allows one class to acquire the properties and behavior of another class, promoting code reuse.
- **Polymorphism** enables the same function or method to behave differently based on the object or context.

These characteristics make OOP flexible, modular, maintainable, secure, and highly suitable for designing complex and large-scale software systems.

Comparison Between Functional Programming and OOP Approach

Point	Functional Programming	OOP Approach
1. Core Concept	Based on mathematical functions and focuses on *what to solve*.	Based on real-world objects and focuses on *how to solve*.
2. Primary Building Block	Uses **pure functions** as the main program unit.	Uses **objects and classes** as the main program unit.
3. Data Handling	Data is **immutable** and cannot be changed once created.	Data is **mutable** and can be modified during execution.
4. State Management	Avoids changing or storing program state.	State is stored and modified inside objects.

5. Approach Style	Follows a **declarative style**, focusing on logic rather than steps.	Follows an **imperative style**, where step-by-step instructions are written.
6. Function Behavior	Uses **pure functions** that do not produce side effects.	Methods may produce **side effects**, such as modifying object data.
7. Reusability Method	Achieved through **higher-order functions and recursion**.	Achieved through **inheritance, polymorphism, and encapsulation**.
8. Iteration Technique	Mostly uses **recursion** instead of loops.	Uses **loops and method calls** for iteration.
9. Relationship Between Data and Behavior	Data and functions are **separate**.	Data and functions are **combined inside objects**.
10. Examples of Languages	Haskell, Scala, Lisp, Erlang.	Java, C++, Python, C#, Ruby.

Overall, functional programming emphasizes **functions, immutability, and mathematical logic**, while OOP emphasizes **objects, reusability, and real-world modeling**. Both approaches have their benefits, and the choice depends on application requirements and development style.

Characteristics of Object-Oriented Languages

Object-oriented languages support several core characteristics that help structure programs using real-world modeling. These characteristics enhance modularity, reusability, security, scalability, and maintainability of software systems. The major characteristics include **Objects, Classes, Inheritance, Reusability, User-Defined Data Types, Polymorphism, and Overloading.**

1. Objects

An object is the basic unit of Object-Oriented Programming. It represents a real-world entity and contains both **data (attributes)** and **behavior (methods)**. Each object has a defined identity, state, and behavior and interacts with other objects to perform operations.

Code Example:

```
class Car {
public:
    string brand;
    void start() {
        cout << "Car Started" << endl;
    }
};

int main() {
    Car c;
```

```
    c.brand = "Audi";
    c.start();
}
```

2. Classes

A class is a blueprint, model, or template used to create objects. It defines data members and member functions without allocating memory until an object is instantiated.

Code Example:

```
class Student {
public:
    string name;
    int roll;

    void show() {
        cout << "Name: " << name << ", Roll: " << roll << endl;
    }
};
```

3. Inheritance

Inheritance allows one class (derived class) to access the properties and methods of another class (base class). It promotes hierarchical relationships and extends functionality without rewriting code.

Types of Inheritance:

Type	Description
Single Inheritance	One derived class inherits from one base class.
Multiple Inheritance	One derived class inherits from more than one base class.
Multilevel Inheritance	A class is derived from another derived class.
Hierarchical Inheritance	Multiple classes inherit from a single base class.
Hybrid Inheritance	Combination of two or more inheritance types.

Code Example (Single):

```
class Animal {
public:
    void sound() {
        cout << "Animal makes sound" << endl;
    }
};

class Dog : public Animal {
public:
    void bark() {
        cout << "Dog barks" << endl;
    }
};
```

4. Reusability

Reusability refers to the capability of using existing code components in new applications. It is primarily achieved through **inheritance, class structures, function reuse, and libraries**, reducing development time and redundancy.

Code Example:

```
class Shape {
public:
    void type() { cout << "This is a Shape" << endl; }
};

class Circle : public Shape { };
class Square : public Shape { };
```

5. User-Defined Data Types

Object-oriented languages allow developers to create customized data structures using classes. These user-defined data types behave similarly to primitive types and are used to represent complex entities.

Code Example:

```
class Employee {
public:
    string name;
    float salary;
};

Employee e1; // user-defined data type
```

6. Polymorphism

Polymorphism means "many forms." It allows the same method name or behavior to act differently depending on the situation. Polymorphism increases flexibility and dynamic behavior in software systems.

Types of Polymorphism:

Type	Description
Compile-Time Polymorphism	Resolved during compilation (method and operator overloading).
Runtime Polymorphism	Resolved at runtime using function overriding and dynamic binding.

Runtime Example:

```
class Animal {
public:
    virtual void sound() { cout << "Animal sound" << endl; }
};

class Cat : public Animal {
public:
    void sound() override { cout << "Meow" << endl; }
```

```
};
```

7. Overloading

Overloading is a form of compile-time polymorphism where a function or operator has the same name but different signatures (parameters).

Sub-Types of Overloading:

Type	Description
Function Overloading	Multiple functions with the same name but different parameters.
Operator Overloading	Redefining operators for custom behavior with user-defined types.

Function Overloading Example:

```
class Math {
public:
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
};
```

Operator Overloading Example:

```
class Number {
public:
    int value;
    Number(int v) { value = v; }

    Number operator+(Number obj) {
        return Number(value + obj.value);
    }
};
```

Unit-2

Introduction to C++:

Identifier and keywords, constants, C++ operators, type conversion, Variable declaration, statements, expressions, input and output, conditional expression loop statements, break control statements, Classes, member functions, objects, arrays of class objects, pointers and classes, nested classes, constructors, destructors Inline member functions, static class member, friend functions, and dynamic memory allocation.

Introduction to C++

C++ is a high-level, general-purpose, object-oriented programming language developed by **Bjarne Stroustrup** in **1979** at AT&T Bell Labs. It was created as an extension of the C language and was originally called **"C with Classes"** before being officially named **C++** in 1983.

C++ supports multiple programming paradigms, including **procedural**, **object-oriented**, and **generic programming**, making it a versatile language. It provides features such as classes, objects, inheritance, polymorphism, templates, and memory management using pointers.

Due to its speed, flexibility, and system-level capabilities, C++ is widely used in applications like operating systems, game development, compilers, embedded systems, and high-performance software.

Identifier and keywords, constants

1. Identifiers

Identifiers are the **names given to variables, functions, classes, arrays, objects, and other user-defined elements** in a program. They help uniquely identify program components. Identifiers are created by the programmer.

Rules for Identifiers:

- Can contain **letters, digits, and underscores**
- Must **not begin with a digit**
- No special characters allowed (except _)
- Case-sensitive (e.g., Age and age are different)
- Cannot be the same as a C++ keyword

Valid Identifier Examples:

name, studentName, total_marks, _amount, num1

Invalid Identifier Examples:

1value, total-marks, class, roll#no

2. Keywords

Keywords are **reserved words** in C++ that have a predefined meaning and purpose in the language. They cannot be used as identifiers (variable or function names).

Examples of C++ Keywords:

int, float, char, class, return, public, private, if, else, while, for, void, new, delete
Example Program:

```
int age = 20;
return 0;
```
Here, int and return are keywords.

3. Constants

Constants are **fixed values** that do not change during program execution. They are used to store data that remains the same throughout the program.

Types of Constants in C++:

Type	Example
Integer Constant	10, -5, 1000
Floating Constant	3.14, -0.55
Character Constant	'A', '3', '#'
String Constant	"Hello", "C++ Programming"
Boolean Constant	true, false

Declaring Constants Using const:

```
const int MAX = 100;
```
Declaring Constants Using #define:

```
#define PI 3.14
```

Example Program Showing All Three

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

#define PI 3.14 // Constant using macro

int main() {
    const int maxStudents = 50; // Constant using const keyword
    int studentCount = 10; // Identifier

    cout << "Max Students: " << maxStudents << endl;
    cout << "Value of PI: " << PI << endl;

    return 0; // 'return' and 'int' are keywords
}
```

C++ Operators

Operators in C++ are special symbols used to perform operations on variables and values. They help in performing calculations, comparisons, logical decisions, memory operations, and many other tasks in a program. Operators work with operands and return a result.

Example:

int a = 10 + 5;
Here, + is an operator and 10 and 5 are operands.

Categories of Operators in C++

C++ operators are classified into the following major types:

1. Arithmetic Operators

Used for performing mathematical operations.

Operator	Meaning	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

Example:

```
int a = 10, b = 3;
cout << a + b; // Output: 13
cout << a % b; // Output: 1
```

2. Relational (Comparison) Operators

Used to compare two values. The result is either true or false.

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example: int x = 5, y = 10;

cout << (x < y); // Output: 1 (true)

3. Logical Operators

Used to combine conditions.

Operator	Meaning	Description
&&	Logical AND	Returns true only if both conditions are true
||	Logical OR	Returns true if at least one condition is true
!	Logical NOT	Reverses the logical value (true → false, false → true)

Example:

```
int a = 5, b = 10;
cout << (a < b && b > 0); // Output: true
```

4. Assignment Operators

Used to assign values to variables.

Operator	Meaning
=	Assign
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign

Example:

```
int a = 10;
a += 5; // a = a + 5 → 15
```

5. Increment and Decrement Operators

Used to increase or decrease value by 1.

Operator	Meaning
++	Increment
--	Decrement

Types:

- **Prefix:** ++a
- **Postfix:** a++

Example:

```
int a = 5;
cout << a++; // Output: 5
cout << ++a; // Output: 7
```

6. Bitwise Operators

Used to perform bit-level operations.

Operator	Name	Description
&	Bitwise AND	Compares each bit of two numbers and returns 1 only if both bits are 1, otherwise 0.
|	Bitwise OR	Returns 1 if at least one bit is 1.
^	Bitwise XOR	Returns 1 if corresponding bits are different; returns 0 if bits are the same.
~	Bitwise NOT	Inverts all bits: 1 becomes 0, and 0 becomes 1.
<<	Left Shift	Shifts bits to the left; each shift multiplies the number by 2.
>>	Right Shift	Shifts bits to the right; each shift divides the number by 2.

Example:

```
int a = 5, b = 3;
cout << (a & b); // Output: 1
```

7. Ternary Operator

Used as shorthand for if-else.

Operator	Syntax
?:	condition ? expression1 : expression2

Example:

```
int a = 10;
string result = (a > 5) ? "Yes" : "No";
```

8. Miscellaneous Operators

Operator	Meaning	Example
sizeof	Returns size of variable or data type	sizeof(int)
&	Address of variable	&a
*	Pointer dereference	*ptr
->	Member access via pointer	ptr->value
.	Member access	obj.value

Example:

```
int a = 10;
cout << sizeof(a); // Output: 4 (depends on system)
```

Example Program Using Multiple Operators

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

int main() {
    int a = 10, b = 5;

    cout << "Arithmetic: " << a + b << endl;
    cout << "Comparison: " << (a > b) << endl;
    cout << "Logical: " << (a > b && b > 0) << endl;

    a += 5;
    cout << "After Assignment: " << a << endl;

    return 0;
}
```

Type Conversion in C++

Type conversion in C++ refers to converting one data type into another. It is required when values of different data types are assigned, used in expressions, or passed to functions. Type conversion helps avoid type mismatch errors and ensures that operations are performed correctly.

Type conversion in C++ is categorized into **two main types**:

1. **Implicit Type Conversion (Automatic Conversion)**
2. **Explicit Type Conversion (Type Casting)**

1. Implicit Type Conversion (Automatic Conversion)

Implicit type conversion is automatically performed by the compiler when values of one data type are assigned to another compatible data type. This is also called **type promotion**.

The conversion usually occurs in the following order (lower → higher precision):

char → int → float → double

Example:

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

int main() {
    int x = 10;
    double y = x; // int automatically converted to double

    cout << "Value of y: " << y; // Output: 10.0
    return 0;
}
```

In this example, the integer value 10 is automatically converted to 10.0 (double).

2. Explicit Type Conversion (Type Casting)

Explicit type conversion is performed manually by the programmer when the compiler does not automatically convert the value or when precise control of conversion is required. This is also known as **type casting**.

Syntax:

```
(type) variable;
```

Example:

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

int main() {
    double num = 10.75;
    int value = (int)num; // manual type casting

    cout << "Value after casting: " << value; // Output: 10
    return 0;
}
```

Forms of Explicit Type Casting in C++

C++ supports multiple casting styles:

Type of Cast	Example	Description
C-Style Cast	(int)x	Traditional cast used in C
Function-Style Cast	int(x)	Uses constructor-style syntax
static_cast	static_cast<int>(x)	Safe cast used in OOP programming

Example Demonstrating All Cast Types:

```
double num = 9.78;

int a = (int)num;               // C-style cast
int b = int(num);               // Function-style cast
int c = static_cast<int>(num); // Modern C++ cast
```

Type Promotion in Expressions

During arithmetic operations, smaller data types are promoted to larger types for accurate results.

Example:

```
char a = 5;
int b = 10;
float c = a + b; // char → int → float

cout << c; // Output: 15.0
```

Variable Declaration, Statements, Expressions, Input and Output in C++

1. Variable Declaration

A variable declaration in C++ is the process of defining a variable with a specific data type and name so it can store a value during program execution. A variable must be declared before it is used.

Syntax: data_type variable_name;

Examples:

```
int age;
float salary;
char grade;
```

A variable declaration may also include initialization:

```
int marks = 90;
```

2. Statements

A statement is a complete instruction that tells the compiler to perform a specific action. Each statement in C++ ends with a semicolon (;). Statements control the logic and execution flow of a program.

Types of Statements:

Type of Statement	Example
Assignment Statement	x = 10;
Input/Output Statement	cin >> x; cout << x;
Control Statement	if (x > 0) {...}
Looping Statement	for(int i=0; i<5; i++) {...}

Example:

```
int a = 5;
```

```
a = a + 2;
```

3. Expressions

An expression in C++ is a combination of variables, constants, and operators that produce a value. Expressions are used in calculations, comparisons, and decision-making.

Types of Expressions:

Type	Description	Example
Arithmetic Expression	Performs mathematical operations	x + y * 2
Relational Expression	Compares values	a > b
Logical Expression	Combines relational expressions	(a > b && b > 0)

Example:

```
int x = 10, y = 5;
int z = x + y * 2; // Expression evaluates to 20
```

4. Input in C++

Input allows the user to enter values during program execution. In C++, input is performed using the cin object from the <iostream> library along with the extraction operator (>>).

Example:

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

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number; // input
}
```

5. Output in C++

Output is used to display information to the user. The cout object with the insertion operator (<<) is used for output operations.

Example:

```
cout << "The number is: " << number;
```

Complete Program Example:

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

int main() {

    int a, b;  // Variable declaration

    cout << "Enter two numbers: "; // Output statement
```

```
    cin >> a >> b; // Input statement

    int result = a + b; // Expression

    cout << "Sum = " << result; // Output statement

    return 0; // Statement
}
```

Summary Table

Concept	Description	Example
Variable Declaration	Defines a variable with type and name	int x;
Statement	Complete instruction ending with ;	x = 5;
Expression	Combination of values and operators	a * b + 10
Input	Accepts values from user	cin >> x;
Output	Displays result to user	cout << x;

Conditional Statements, Loop Statements, break Control Statement

1. Conditional Statements

Conditional statements in C++ are used to make decisions based on conditions. They allow program execution to follow different paths depending on whether a condition evaluates to **true** or **false**.

Types of Conditional Statements:

a) if Statement

Executes a block only if the condition is true.

```
if (age >= 18) {
    cout << "Eligible to vote";
}
```

b) if-else Statement

Provides two paths: one when the condition is true and another when it is false.

```
if (marks >= 40) {
    cout << "Pass";
} else {
    cout << "Fail";
}
```

c) else-if Ladder

Used to check multiple conditions sequentially.

```
if (score >= 90) {
```

```
    cout << "Grade A";
} else if (score >= 75) {
    cout << "Grade B";
} else {
    cout << "Grade C";
}
```

d) switch Statement

Used when multiple values of a single expression need to be tested.

```
int day = 3;
switch(day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    case 3: cout << "Wednesday"; break;
    default: cout << "Invalid day";
}
```

2. Loop Statements

Loop statements allow repeated execution of a block of code as long as a specified condition remains true.

Types of Looping Statements:

a) for Loop

Used when the number of iterations is known.

```
for(int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

b) while Loop

Used when the number of iterations is unknown and depends on a condition.

```
int i = 1;
while(i <= 5) {
    cout << i << " ";
    i++;
}
```

c) do-while Loop

Executes the block at least once, even if the condition is false.

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while(i <= 5);
```

3. Break Control Statement

The break statement is used to immediately exit from a **loop** or a **switch statement**, regardless of the loop condition. When encountered, control transfers to the first statement after the loop or switch block.

Example in Loop:

```
for(int i = 1; i <= 10; i++) {
    if(i == 5) {
        break; // loop terminates when i becomes 5
    }
    cout << i << " ";
}
```

Example in switch:

```
int num = 2;

switch(num) {
    case 1: cout << "One"; break;
    case 2: cout << "Two"; break; // break prevents fall-through
    case 3: cout << "Three"; break;
}
```

Summary Table

Concept	Purpose	Example
Conditional Statements	Decision making based on conditions	if, if-else, switch
Loop Statements	Repeat execution	for, while, do-while
break Statement	Exit loop or switch immediately	break;

Classes

A class in C++ is a user-defined data type that provides a template or blueprint for creating objects. It groups data members and member functions together into a single unit. The class defines the structure and behavior that the objects created from it will possess. Data members represent the attributes of a class, while member functions operate on those attributes. A class does not occupy memory until an object is instantiated from it. Classes support key OOP concepts such as abstraction, encapsulation, inheritance, and polymorphism, making them the foundation of object-oriented programming in C++.

Example:

```
class Student {
public:
    string name;
    int roll;
    void show() {
        cout << "Name: " << name << ", Roll: " << roll;
    }
};
```

Member Functions

Member functions are functions declared inside a class and are responsible for defining the behavior of an object. They operate on the class data members and can access private, protected, or public data within the class. Member functions enforce encapsulation since they provide controlled access to the internal representation of the object. They may be defined inside the class (automatically inline) or outside using the scope resolution operator ::.

Example:

```
class Student {
public:
    string name;
    void display() {
        cout << "Student Name: " << name;
    }
};
```

Objects

An object is an instance of a class and represents a real-world identifiable entity. When an object is created, memory is allocated for storing its data members. Objects interact with each other through member functions, and each object maintains its own separate identity and state, even when multiple objects are created from the same class. Objects allow programs to be modeled more naturally and logically.

Example:

```
Student s1;
s1.name = "Rahul";
s1.display();
```

Arrays of Class Objects

Arrays of class objects allow multiple instances of a class to be stored and accessed sequentially. This is useful when handling a collection of similar objects, such as multiple students, employees, or products. Each element in the array behaves like an independent object with its own state.

Example:

```
Student s[3];
s[0].name = "Amit";
s[1].name = "Sara";
```

Pointers and Classes

Pointers can be used to store the address of an object and access its members using the arrow (->) operator. Using dynamic memory and pointers enables the creation of objects during runtime and supports advanced memory management.

Example:

```
Student *ptr = new Student();
ptr->name = "John";
ptr->display();
```

Nested Classes

A nested class is a class defined within another class. It is useful for logically grouping classes and restricting access. The nested class can access private members of the outer class depending on access specifiers.

Example:

```
class Outer {
public:
    class Inner {
    public:
        void show() {
            cout << "Inner class accessed";
        }
    };
};
```

Constructors

A constructor is a special member function automatically invoked when an object is created. It initializes object data and has the same name as the class with no return type. Constructors support overloading and can be default, parameterized, copy constructors, or dynamic constructors.

Example:

```
class Car {
public:
    string model;
    Car(string m) {
        model = m;
    }
};
```

Destructors

A destructor is a special member function automatically executed when an object goes out of scope or is deleted. It begins with a tilde (~) and has no return type and no parameters. Destructors are used for cleanup tasks such as memory deallocation or closing files.

Example:

```
class Car {
public:
    ~Car() {
        cout << "Destructor executed";
    }
};
```

Inline Member Functions

Inline member functions are expanded at compile time rather than invoked as a normal function call, reducing overhead. Inline is suitable for short functions. Functions defined inside a class are automatically treated as inline unless complexity prevents it.

Example:

```
inline void display() {
    cout << "Inline function executed";
```

```
}
```

Static Class Member

A static data member belongs to the class rather than any individual object. All objects share a single copy of the static member. Static member functions can only access static data.

Example:

```
class Counter {
public:
    static int count;
};
int Counter::count = 0;
```

Friend Functions

A friend function is a non-member function that is granted special access privileges to private and protected data of a class. It helps in operator overloading and two-class communication.

Example:

```
class Box {
private:
    int width;
public:
    Box(int w): width(w) {}
    friend void show(Box b);
};
```

Dynamic Memory Allocation

Dynamic memory allocation in C++ allows the creation of objects and variables at runtime using new and deallocation using delete. It enables flexibility and efficient memory usage.

Example:

```
Student *ptr = new Student();
delete ptr;
```

Unit-3

Polymorphism and Inheritance:

Function overloading, operator overloading, polymorphism, early binding, polymorphism with pointers, virtual functions, late binding, pure virtual functions.
Single inheritance, types of inheritance, types of base classes, types of derivations, multiple inheritances, container classes, member access control.

Polymorphism in C++

Polymorphism is one of the core features of Object-Oriented Programming that allows a function, operator, or object to behave in multiple forms depending on the context. The term polymorphism means *"many forms."* In C++, polymorphism enables different implementations of functions or operations while keeping the same interface. It improves flexibility, scalability, and code reusability. Polymorphism is categorized into **compile-time polymorphism** and **runtime polymorphism**, based on when the function binding occurs.

1. Function Overloading

Function overloading is a form of compile-time polymorphism where multiple functions share the same name but differ in either number, type, or order of parameters. The appropriate function is resolved during compilation.

Example:

```
class Math {
public:
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
};
```

2. Operator Overloading

Operator overloading allows built-in operators to work with user-defined objects. It improves code readability by redefining how operators behave when applied to objects of a class. Operator overloading also belongs to compile-time polymorphism.

Example:

```
class Number {
public:
    int value;
    Number(int v) { value = v; }

    Number operator+(Number obj) {
        return Number(value + obj.value);
    }
};
```

3. Polymorphism (General Concept)

Polymorphism allows a single interface or function name to represent multiple forms or implementations. It enables the same function call to behave differently based on context or data type. In C++, polymorphism is implemented in two ways: **compile-time polymorphism**, achieved through function and operator overloading, and **runtime polymorphism**, achieved through virtual functions.

Example:

```
class Shape {
public:
    void area(int side) {
        cout << "Area of Square: " << side * side << endl;
    }

    void area(int length, int width) {
        cout << "Area of Rectangle: " << length * width << endl;
    }
};

int main() {
    Shape s;
    s.area(5);         // Calls square version
    s.area(4, 6);      // Calls rectangle version
}
```

In this example, the same function name area() performs different tasks depending on the parameters, demonstrating polymorphism.

4. Early Binding

Early binding, also known as **static binding** or **compile-time binding**, occurs when the function call is resolved at compile time. Features such as function overloading and operator overloading follow early binding because the compiler determines which version of the function or operator to execute based on the function signature.

Example:

```
class Display {
public:
    void show(int x) {
        cout << "Integer: " << x << endl;
    }

    void show(string text) {
        cout << "String: " << text << endl;
    }
};

int main() {
    Display d;
    d.show(10);          // Calls show(int)
    d.show("Hello");     // Calls show(string)
}
```

Here, the compiler selects the correct version of show() during compilation, demonstrating early binding.

5. Polymorphism Using Pointers

C++ allows base class pointers to refer to derived class objects. This feature enables polymorphic behavior, especially when used with virtual functions. Without virtual functions, pointer calls follow the pointer type rather than the object type.

Example:

```
class Base {
public:
   void show() { cout << "Base"; }
};

class Derived : public Base {
public:
   void show() { cout << "Derived"; }
};

Base* ptr;
Derived d;
ptr = &d;
ptr->show(); // Calls Base version (No virtual keyword → early binding)
```

6. Virtual Functions

A virtual function is a member function declared using the keyword virtual in the base class to enable dynamic function overriding in derived classes. When accessed through a base class pointer, the function of the actual object type executes—not the pointer type. This supports **runtime polymorphism**.

Example:

```
class Base {
public:
   virtual void show() { cout << "Base"; }
};

class Derived : public Base {
public:
   void show() override { cout << "Derived"; }
};
```

7. Late Binding

Late binding, also known as **runtime binding** or **dynamic binding**, occurs when the function call is resolved at runtime rather than compile time. This mechanism works only when a function is declared as virtual in the base class. Late binding ensures that the overridden method in the derived class executes even if the call is made through a base class pointer.

Example:

```
class Base {
public:
   virtual void message() {
      cout << "Message from Base class" << endl;
   }
};

class Derived : public Base {
public:
   void message() override {
      cout << "Message from Derived class" << endl;
   }
};
```

```
int main() {
    Base* ptr;
    Derived d;
    ptr = &d;
    ptr->message();   // Executes derived class function (runtime decision)
}
```
In this example, the function call is resolved at runtime, enabling polymorphic behavior through late binding.

8. Pure Virtual Functions

A pure virtual function is declared in a base class but does not provide a definition. It is assigned = 0. Any class containing at least one pure virtual function becomes an **abstract class** and cannot be instantiated directly. Pure virtual functions enforce overriding in derived classes.

Example:

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

Inheritance in C++

Inheritance is a fundamental concept in object-oriented programming that allows one class (called the **derived class**) to inherit properties and behaviors (data members and member functions) from another class (called the **base class**). It helps in code reusability, reduces redundancy, supports hierarchical relationships, and allows modifications or extensions of existing code without rewriting it.

1. Single Inheritance

Single inheritance involves one base class and one derived class. It forms a simple one-to-one parent-child relationship.

Example:

```
class Animal {
public:
    void sound() { cout << "Animal sound"; }
};

class Dog : public Animal { };
```

2. Types of Inheritance

C++ supports several types of inheritance based on relationship structure:

a) Single Inheritance

(One base → one derived)

```
class A { };
class B : public A { };
```

b) Multilevel Inheritance

(Base → Derived → Another Derived)

```
class A { };
class B : public A { };
class C : public B { };
```

c) Multiple Inheritance

(A derived class has more than one base class)

```
class A { };
class B { };
class C : public A, public B { };
```

d) Hierarchical Inheritance

(One base → multiple derived classes)

```
class A { };
class B : public A { };
class C : public A { };
```

e) Hybrid Inheritance

(Combination of two or more inheritance types)

```
class A { };
class B : public A { };
class C { };
class D : public B, public C { };
```

3. Types of Base Classes (Based on Access)

Access specifiers define how base class members are inherited.

a) Public Base Class

Public remains public, protected remains protected.

```
class Base {
public:
    int x;
};

class Derived : public Base { };
```

b) Protected Base Class

Public and protected become protected in derived class.

```
class Base {
public:
    int x;
};

class Derived : protected Base { };
```

c) Private Base Class

Public and protected become private in derived class.

```
class Base {
public:
    int x;
};

class Derived : private Base { };
```

4. Types of Derivations

Types of derivations refer to how a derived class inherits members of the base class using different access modes. The accessibility of inherited members in the derived class depends on whether the class is inherited as **public, protected, or private**.

- **Public derivation:** Public members of the base class remain public in the derived class, and protected members remain protected.
- **Protected derivation:** Public and protected members of the base class become protected in the derived class.
- **Private derivation:** Public and protected members of the base class become private in the derived class.

Example:

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

class A {
public:
    void display() {
        cout << "Base Class A" << endl;
    }
};

// Public Derivation
class B : public A { };

// Protected Derivation
class C : protected A { };

// Private Derivation
class D : private A { };

int main() {
    B obj1;
    obj1.display();  // Accessible (public inheritance)

    // C obj2;
    // obj2.display();  // ❌ Not accessible because display() becomes protected

    // D obj3;
    // obj3.display();  // ❌ Not accessible because display() becomes private
}
```

5. Multiple Inheritance

Multiple inheritance occurs when a derived class inherits from more than one base class. It allows combining functionality from multiple sources.

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

class A {
public:
    void displayA() { cout << "From A\n"; }
};

class B {
public:
    void displayB() { cout << "From B\n"; }
};

class C : public A, public B { };

int main() {
    C obj;
    obj.displayA();
    obj.displayB();
}
```

6. Container Classes

A container class stores and manages objects of another class. Such classes act as storage structures like arrays, lists, or user-defined collections.

Example:

```
class Student {
public:
    string name;
};

class Classroom {
private:
    Student list[30]; // container storing objects
};
```

7. Member Access Control

Member access control determines which class members are visible and accessible in derived classes. C++ uses three access levels:

- **Public:** Accessible everywhere
- **Protected:** Accessible in base and derived class
- **Private:** Accessible only within the class

Example:

```
class Base {
protected:
    int value;
```

```
};

class Derived : public Base {
public:
    void show() {
        cout << value; // allowed because value is protected
    }
};
```

Unit-4

Exceptions and Templates:

Exception Syntax, Multiple Exceptions, Function Templates, Function Templates with multiple argument templates.

Exceptions and Templates in C++

C++ provides two powerful programming features: **Exception Handling** and **Templates**. Exception handling enables safe run-time error management without abrupt program termination, while templates allow writing generic and reusable code that works with different data types. Both concepts improve program reliability, flexibility, and maintainability.

Exception Handling in C++

Exception handling is used to handle abnormal runtime conditions or errors in a controlled manner. Without exception handling, runtime errors such as division by zero, invalid memory access, or file-handling failures may terminate a program unexpectedly.

Exception handling uses three keywords:

- try → Represents code that may throw an exception
- throw → Used to raise an exception
- catch → Handles the thrown exception

Example using all three keywords:

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

int main() {
    int a, b;

    cout << "Enter two numbers: ";
    cin >> a >> b;

    try {
        if(b == 0) {
            throw "Error: Division by zero!";
        }
        cout << "Result: " << a / b << endl;
    }
    catch(const char* message) {
        cout << message << endl;
    }

    cout << "Program continues..." << endl;

    return 0;
}
```

Explanation:

- The **try block** contains the division operation that may cause an error.

- When the second number (b) is **zero**, the program executes a **throw statement**, sending an exception message.
- The thrown message is received and handled inside the **catch block**, preventing program termination.

1. Exception Syntax

The basic syntax of exception handling involves writing code that may fail inside a try block, and handling errors using one or more catch blocks. A throw statement transfers control to the appropriate handler.

Example:

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

int main() {
   try {
      int x = 0;
      if(x == 0)
         throw "Cannot divide by zero!";
      cout << 10 / x;
   }
   catch(const char* err) {
      cout << "Exception: " << err << endl;
   }
}
```

2. Multiple Exceptions

A single try block may generate different types of exceptions. C++ allows multiple catch blocks, each designed to handle a specific type of exception. The first matching handler executes.

Example:

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

int main() {
   try {
      int input = 2;

      if(input == 1)
         throw 100;         // integer exception
      else if(input == 2)
         throw 5.5;         // double exception
      else
         throw string("Unknown Error");
   }
   catch(int e) {
      cout << "Integer Exception: " << e << endl;
   }
   catch(double e) {
      cout << "Double Exception: " << e << endl;
   }
   catch(string s) {
      cout << "String Exception: " << s << endl;
```

```
    }
}
```

3. Catch-All Handler

C++ also provides a **generic catch block** that catches any exception using catch(...).

```
catch(...) {
    cout << "Unhandled exception occurred.";
}
```

Templates in C++

Templates enable generic programming, allowing functions and classes to operate with different data types without rewriting code. Templates are defined using the template keyword.

There are two major types:

- **Function Templates**
- **Class Templates** (not requested but optional concept context)

Templates promote code reusability, reduce duplication, and support type flexibility.

4. Function Templates

A function template is a template for creating multiple function versions using different data types. The compiler generates the required function during compilation, depending on the argument type used.

Syntax:

```
template <typename T>
return_type function_name(T parameter) { … }
```

Example:

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

template <class T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maximum(10, 20) << endl;     // int
    cout << maximum(3.5, 2.1) << endl;   // double
}
```

5. Function Templates with Multiple Argument Templates

Function templates in C++ can accept more than one type parameter. This allows the same function to work with arguments of different data types without creating separate versions. By defining multiple template parameters, the function becomes more flexible and capable of handling different data type combinations.

Example:

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

template <typename T1, typename T2>
void display(T1 a, T2 b) {
    cout << "Value 1: " << a << ", Value 2: " << b << endl;
}

int main() {
    display(5, 3.14);         // Uses int and double
    display("Hello", 100);    // Uses string and int
}
```

6. Template Specialization (Related Concept)

When a template requires different behavior for a specific data type, C++ allows specialization.

Example:

```
template <>
string maximum<string>(string a, string b) {
    return (a.length() > b.length()) ? a : b;
}
```

Unit-5

File Handling in C++:

C++ Streams, Console Stream Classes, Formatted And Unformatted Console I/O Operations, manipulators, File Streams, Classes File Modes, File Pointers and Manipulations File I/O.

File Handling in C++

File handling in C++ allows a program to store data permanently on storage devices rather than keeping it only in memory. This makes the program more useful because the stored data can be reused later. File handling supports operations such as creating files, writing data, reading data, updating records, and deleting records. C++ provides this functionality through the <fstream> library.

1. C++ Streams

A **stream** in C++ is a flow of data between a program and an input/output device. When the program reads input, data flows from the device *into* the program (input stream). When printing output, data flows *from* the program to the device (output stream).

C++ treats files and console input/output the same way using streams, making operations consistent and easy to use.

Example:

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

int main() {
    int num;
    cout << "Enter number: ";   // Output stream
    cin >> num;                 // Input stream
    cout << "You entered: " << num;
}
```

2. Console Stream Classes

C++ provides built-in classes to handle console input/output through streams. These classes are available in the <iostream> header.

Class	Purpose	Associated Object
istream	Handles input operations	cin
ostream	Handles output operations	cout
iostream	Handles both input and output	Used in advanced cases

These classes overload operators (>> and <<) to simplify input/output.

Example:

```
cout << "Hello";   // uses ostream
cin >> num;        // uses istream
```

3. Formatted Console I/O Operations

Formatted I/O allows controlling the appearance of output. Using **manipulators** (like setw, fixed, setprecision, etc.) we can control alignment, width, decimal formatting, and spacing of displayed values.

Formatted I/O improves output readability and presentation.

Example:

```
#include <iomanip>
cout << setw(10) << setprecision(2) << fixed << 45.6789;
```

Output:

```
    45.68
```

4. Unformatted Console I/O Operations

Unformatted I/O handles raw character input/output and does not apply formatting. It is faster and is used when raw character processing is required (like reading a file word-by-word or character-by-character).

Common unformatted functions:

- get() → reads a single character
- put() → prints a single character
- getline() → reads full string including spaces

Example:

```
char ch;
cin.get(ch);
cout.put(ch);
```

5. Manipulators

Manipulators are special functions used with streams to modify how data is displayed or read. They help format output without changing variable values.

Two types exist:

- **Without arguments** → endl, hex, oct, dec
- **With arguments** → setw(n), setprecision(n), setfill(n)

Example:

```
cout << hex << 255;      // output: ff (hex form)
```

6. File Streams

C++ uses three classes from <fstream> to work with files:

Class	Purpose
ifstream	Read data from file
ofstream	Write data into file
fstream	Read and write both

These streams work similarly to console streams but interact with files.

Example: Writing and Reading

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

int main() {
    ofstream out("test.txt");
    out << "Hello File!";
    out.close();
}
```

7. File Modes

File modes define how a file should be opened. Multiple modes can be combined using |.

Mode	Meaning
ios::in	Open file for reading
ios::out	Open file for writing
ios::app	Append new contents to end
ios::trunc	Delete old content
ios::binary	Open file in binary format

Example:

```
fstream file;
file.open("data.txt", ios::out | ios::app);
file << "New line added.";
file.close();
```

8. File Pointers and Manipulation

File pointers are used to track the current position in the file for reading or writing. There are two pointers:

- **get pointer** → used to read from file (seekg(), tellg())
- **put pointer** → used to write to file (seekp(), tellp())

These allow random access — meaning we can jump to any position inside file.

Example:

```
fstream file("data.txt");
file.seekg(0);                    // Move read pointer to start
cout << file.tellg();             // Show pointer position
```

9. File I/O (Read and Write Operations)

File Input/Output (I/O) refers to writing data to a file and reading it back.

Writing Example:

```
ofstream file("info.txt");
file << "This is a test file.";
file.close();
```

Reading Example:

```
ifstream file("info.txt");
string text;
while(getline(file, text)) {
    cout << text << endl;
}
file.close();
```

Binary File Example (Advanced but important)

```
struct Student {
    char name[20];
    int age;
};

Student s = {"John", 20};

ofstream fout("student.dat", ios::binary);
fout.write((char*)&s, sizeof(s));
fout.close();
```

1. C++ Streams (cin, cout)

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

int main() {
    int age;

    cout << "Enter your age: ";   // Output stream (cout)
    cin >> age;                  // Input stream (cin)

    cout << "You entered: " << age << endl;

    return 0;
}
```

2. Console Stream Classes (istream, ostream)

You usually don't use istream and ostream directly; instead you use their objects: cin and cout.

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

int main() {
    int x;
    cout << "Enter a number: ";   // ostream object
    cin >> x;                    // istream object

    cout << "Number is: " << x << endl;

    return 0;
}
```

3. Formatted Console I/O Operations

Using <iomanip> manipulators like setw, setprecision, fixed.

```
#include <iostream>
#include <iomanip>      // for setw, setprecision, fixed
using namespace std;

int main() {
    double price = 123.456789;

    cout << "Default: " << price << endl;

    cout << "Fixed, 2 decimals: "
         << fixed << setprecision(2) << price << endl;

    cout << "Width 10: " << setw(10) << price << endl;

    return 0;
}
```

4. Unformatted Console I/O Operations (get, getline, put)

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

int main() {
    char ch;
    char name[50];

    cout << "Enter a single character: ";
    cin.get(ch);             // reads one character
    cout << "You entered: ";
    cout.put(ch);            // prints one character
    cout << endl;

    cin.ignore();            // clear leftover newline

    cout << "Enter your full name: ";
    cin.getline(name, 50);   // reads a full line including spaces

    cout << "Your name is: " << name << endl;

    return 0;
}
```

5. Manipulators (endl, hex, dec, oct, setw, setprecision)

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

int main() {
    int num = 255;
    double val = 12.34567;

    // Number system manipulators
    cout << "Decimal: " << dec << num << endl;
    cout << "Hex: "     << hex << num << endl;
    cout << "Octal: "   << oct << num << endl;

    // Formatting floating number
    cout << fixed << setprecision(2);
    cout << "Fixed with 2 decimals: " << val << endl;

    // Width using setw
    cout << "Right aligned in 10 spaces: "
         << setw(10) << val << endl;

    cout << "End of program." << endl;  // endl = newline + flush

    return 0;
}
```

6. File Streams (ifstream, ofstream, fstream)

(a) Write to a file using ofstream

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

int main() {
    ofstream outFile("example.txt");   // open file for writing

    if (!outFile) {
        cout << "File could not be opened!" << endl;
        return 1;
    }

    outFile << "Hello from C++ file handling!" << endl;
    outFile << "This is a second line." << endl;

    outFile.close();   // close the file

    cout << "Data written to example.txt" << endl;
    return 0;
}
```

(b) Read from a file using ifstream

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

int main() {
    ifstream inFile("example.txt");   // open file for reading

    if (!inFile) {
        cout << "File not found!" << endl;
        return 1;
    }

    string line;
    while (getline(inFile, line)) {   // read line by line
        cout << line << endl;
    }

    inFile.close();
    return 0;
}
```

(c) Read & Write using fstream

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

int main() {
    fstream file("data.txt", ios::in | ios::out | ios::trunc);

    if (!file) {
```

```
        cout << "File error!" << endl;
        return 1;
    }

    file << "First line" << endl;
    file << "Second line" << endl;

    file.seekg(0);   // move read pointer to beginning

    string line;
    while (getline(file, line)) {
        cout << line << endl;
    }

    file.close();
    return 0;
}
```

7. File Modes (ios::in, ios::out, ios::app, ios::binary, ios::trunc)

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

int main() {
    // Open in append mode (add at end)
    ofstream appFile("log.txt", ios::app);
    appFile << "New log entry." << endl;
    appFile.close();

    // Open in write mode with truncation (old data removed)
    ofstream writeFile("log.txt", ios::out | ios::trunc);
    writeFile << "File overwritten." << endl;
    writeFile.close();

    // Open in binary mode
    ofstream binFile("data.bin", ios::binary);
    int x = 100;
    binFile.write((char*)&x, sizeof(x));
    binFile.close();

    cout << "File modes demo completed." << endl;
    return 0;
}
```

8. File Pointers and Manipulations (seekg, seekp, tellg, tellp)

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

int main() {
    ofstream out("sample.txt");
    out << "Hello file pointer!";
    out.close();

    fstream file("sample.txt", ios::in | ios::out);

    if (!file) {
        cout << "File error!" << endl;
```

```
        return 1;
    }

    // Show current get pointer position
    cout << "Initial get position: " << file.tellg() << endl;

    // Move get pointer to 6th character
    file.seekg(6);
    cout << "New get position: " << file.tellg() << endl;

    char ch;
    file.get(ch);   // read one char at current position
    cout << "Character at position 6: " << ch << endl;

    // Move put pointer to end and write text
    file.seekp(0, ios::end);
    file << " [Appended]";

    file.close();
    return 0;
}
```

9. File I/O – Full Simple Example (Write + Read)

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

int main() {
    // Writing to file
    ofstream out("students.txt");
    if (!out) {
        cout << "Cannot open file for writing." << endl;
        return 1;
    }

    out << "Alice 20" << endl;
    out << "Bob 21" << endl;
    out.close();

    // Reading from file
    ifstream in("students.txt");
    if (!in) {
        cout << "Cannot open file for reading." << endl;
        return 1;
    }

    string name;
    int age;

    cout << "Student Records:" << endl;
    while (in >> name >> age) {  // read name and age
        cout << "Name: " << name << ", Age: " << age << endl;
    }

    in.close();
    return 0;
}
```