STUDY MATERIAL FOR B.Sc., COMPUTER SCIENCE WITH ARTIFICIAL INTELLIGENCE

C++ - PROGRAMMING

SEMESTER – II

ACADEMIC YEAR 2025-26

PREPARED BY

COMPUTER SCIENCE DEPARTMENT

INDEX

UNIT - I

Introduction to C++ and Object-Oriented Programming (OOP)

C++ is a hybrid language that supports both procedural and object-oriented paradigms. In 2026, it remains a cornerstone for system software and performance-critical applications due to its efficiency.

Key Concepts of OOP

Objects: Basic run-time entities representing real-world items (e.g., a person or bank account) that contain both data and code.

Classes: User-defined blueprints or prototypes used to create objects; they bind data and member functions together.

Encapsulation: Wrapping data and functions into a single unit (class) to restrict direct access, often called "data hiding".

Abstraction: Showing only essential features while hiding background details to reduce complexity.

Inheritance: The mechanism by which a new class (subclass) acquires properties of an existing class (superclass), promoting code reusability.

Polymorphism: The ability of a single interface to behave differently based on the object type (e.g., a drive() method working differently for a Car vs. a Truck).

Advantages of OOP

Modularity: Programs are divided into independent modules, making them easier to manage, debug, and maintain.

Reusability: Inheritance and classes allow developers to reuse code across projects, reducing redundancy (DRY principle).

Security: Data hiding ensures that internal object states cannot be tampered with by external functions.

Scalability: Systems can be easily upgraded from small to large by adding new objects or inheriting from base classes.

Object-Oriented Languages

C++ is often categorized as a "middle-level" or "hybrid" language. Other prominent OOP languages include Java, Python, Smalltalk, and Ada.

I/O and Declarations in C++

I/O Streams: C++ uses streams (sequences of bytes) for data flow. The <iostream> header defines standard objects:

cin: Standard input (keyboard), used with the extraction operator >>.

cout: Standard output (screen), used with the insertion operator <<.

cerr: Standard error, unbuffered output.

Declarations: Variables must be declared with a type (e.g., int len;) before use. C++ is type-safe, meaning the compiler checks for operations defined for that specific type.

Control Structures

Decision Making and Statements

If...else: Executes a block if a condition is true; else provides a fallback if false.

Switch case: Efficiently selects one block from many constant-value options based on a single variable's value.

Jump Statements:

break: Immediately exits the current loop or switch block.

continue: Skips the current iteration and moves to the next one.

goto: Transfers control to a labeled statement (generally discouraged for maintainability).

Loops in C++

for: Best when the number of iterations is known beforehand; contains initialization, condition, and update in one line.

while: Repeats as long as a condition is true; used when the iteration count is unknown.

do-while: Similar to while, but guarantees the body executes at least once before checking the condition.

Functions in C++

Functions: Blocks of code designed to perform specific tasks, promoting modularity and reuse.

Inline Functions:

Definition: Small functions declared with the inline keyword. The compiler replaces the call with the actual code at compile time.

Advantage: Reduces function call overhead (saving time on stack operations like push/pop).

Limitation: Can increase binary size ("code bloat") if overused; compilers may ignore the request for complex functions (loops, recursion).

Function Overloading:

Concept: Defining multiple functions with the same name but different parameters (number, type, or both).

Mechanism: The compiler selects the correct function based on the arguments provided during the call (compile-time polymorphism).

Rule: Functions cannot be overloaded based on return type alone; they must differ in their parameter list.

```
#include <iostream>
#include <string>
using namespace std;
// 1. BASE CLASS (Abstraction & Encapsulation)
class Product {
protected:
   string name;
   double price;
public:
   // Constructor
   Product(string n, double p) : name(n), price(p) {}

   // 2. INLINE FUNCTION: Small, frequent operation
   inline double getPrice() const { return price; }

   // 3. VIRTUAL FUNCTION: Basis for Runtime Polymorphism
   virtual void displayDetails() {
      cout << "Product: " << name << " | Price: $" << price << endl;
   }
```

```
};

// 4. DERIVED CLASS (Inheritance)
class Electronics : public Product {
private:
    int warrantyMonths;

public:
    Electronics(string n, double p, int w) : Product(n, p), warrantyMonths(w) {}

    // Overriding virtual function (Runtime Polymorphism)
    void displayDetails() override {
        cout << "[Electronics] " << name << " | Price: $" << price
             << " | Warranty: " << warrantyMonths << " months" << endl;
    }
    // 5. FUNCTION OVERLOADING: Same name, different parameters
    void applyDiscount(double percentage) {
        price -= (price * (percentage / 100));
        cout << "Discount applied. New price: $" << price << endl;
    }
    void applyDiscount(int flatAmount) {
        price -= flatAmount;
        cout << "Flat discount applied. New price: $" << price << endl;
    }
};
int main() {
    // 6. DECLARATIONS & I/O
    int choice;
```

```
Electronics laptop("Gaming Laptop", 1200.0, 24);
Product* ptr = &laptop; // Base class pointer to derived object

cout << "--- Store Management System 2026 ---" << endl;

// 7. LOOPS (do-while)
do {
    cout << "\n1. View Details\n2. Apply Percentage Discount\n3. Apply Flat Discount\n4. Exit\nEnter Choice: ";
    cin >> choice;
    // 8. DECISION MAKING (Switch Case)
    switch (choice) {
        case 1:
            ptr->displayDetails(); // Polymorphism in action
            break;
        case 2: {
            double p;
            cout << "Enter percentage: "; cin >> p;
            laptop.applyDiscount(p); // Overloaded function call
            break;
        }
        case 3: {
            int flat;
            cout << "Enter flat amount: "; cin >> flat;
            laptop.applyDiscount(flat); // Overloaded function call
            break;
        }
        case 4:
```

```
            cout << "Exiting..." << endl;
            break;
        default:
            // 9. IF...ELSE & JUMP (continue)
            if (choice < 1 || choice > 4) {
                cout << "Invalid Option!" << endl;
                continue;
            }
    }
            // 10. FOR LOOP: Just for demonstration
    if(choice == 1) {
        cout << "Rating: ";
        for(int i=0; i<5; i++) cout << "*";
        cout << endl;
    }

  } while (choice != 4);
  return 0;
}
```

Key Integration Points in this Code:

Encapsulation: The Product and Electronics classes bundle data (name, price) with methods that operate on them.

Inheritance: Electronics inherits from Product, reusing its members and constructor.

Polymorphism: ptr->display Details() uses a base class pointer to call a derived class method at runtime.

Function Overloading: Two versions of apply Discount exist, distinguished by their parameter types (double vs. int).

Inline Function: get Price() is marked inline to suggest the compiler replace the call with the actual code to save overhead.

Control Structures: A do-while loop maintains the menu, a switch handles user selection, and if-else validates input

UNIT - II

Classes & Objects

Declaring Objects

A class is a user-defined blueprint, while an object is a specific instance that occupies memory.

Syntax: ClassName objectName;

Memory: No memory is allocated when the class is defined; it is allocated only when the object is instantiated.

Access: Members are accessed using the dot operator (.) for local objects or the arrow operator (->) for pointers.

Understanding object declaration is critical because defining a class allocates no memory; memory is only reserved when an object is instantiated.

1. Conceptual Overview

Class: A user-defined data type that acts as a template, specifying attributes (data) and behaviors (functions).

Object: A physical entity or "instance" created from a class blueprint.

Instantiation: The process of declaring an object, which triggers memory allocation and optional initialization via a constructor.

2. Methods of Declaring Objects

You can declare objects in several ways depending on how you intend to manage their memory and lifetime.

A. Automatic (Stack) Declaration

This is the most common method, where the object is created on the stack memory. It is automatically destroyed when it goes out of scope.

Syntax: ClassName objectName;

Example:

```
class Room {
public:
   double length, breadth;
};
int main() {
```

```
    Room room1; // Object declared on the stack
    room1.length = 10.5;
    return 0;
}
```

B. Dynamic (Heap) Declaration

Used when you need an object to persist beyond its current scope or when the object's size is large. You must manually manage this memory using new and delete.

Syntax: ClassName pointerName = new ClassName();*

Example:

```
Room* myRoomPtr = new Room(); // Object created on the heap
myRoomPtr->length = 25.0;     // Access via arrow operator
delete myRoomPtr;             // Manual deallocation required
```

C. Immediate Declaration (at Class Definition)

C++ allows you to declare objects immediately after the class body, before the terminating semicolon.

Example:

```
class Point {
    int x, y;
} p1, p2; // p1 and p2 are declared immediately
```

3. Object Lifetime and Access

Global Objects: Declared outside all functions; they are accessible throughout the entire program and live for its duration.

Local Objects: Declared inside a function or block; accessible only within that scope.

Member Access: Use the dot operator (.) for stack objects and the arrow operator (->) for pointers to heap objects.

Defining Member Functions

Member functions provide the interface for manipulating a class's data. They can be defined in two locations:

Inside the Class Definition:

Functions defined directly within the class body are implicitly treated as inline.

Best Use: For small, simple functions like getters or setters to reduce function call overhead.

Outside the Class Definition:

The function is declared inside the class but defined elsewhere using the scope resolution operator (::).

Best Use: For large or complex functions to keep the class definition readable and organized.

Syntax: return_type ClassName::FunctionName() { ... }.

Example: Member Function Definitions

```
class Box {
public:
   double length;
   // Defined inside (Implicitly inline)
   double getArea() { return length * length; }

   // Declared inside, defined outside
   void setLength(double len);
};
// Definition outside using scope resolution
void Box::setLength(double len) {
    length = len;
}
```

2. Static Member Variables

Static member variables are class-level variables shared by all instances.

Shared Memory: Only one copy exists in memory, regardless of how many objects are created.

Lifetime: They persist for the entire duration of the program.

Initialization: Must be defined and initialized outside the class (usually in a .cpp file) to allocate memory. If not explicitly initialized, they default to zero.

3. Static Member Functions

These functions are associated with the class itself rather than any specific object instance.

No this Pointer: Because they do not belong to a specific object, they lack the implicit this pointer.

Access Restrictions: They can only access other static member variables or static member functions. They cannot access non-static data directly.

Calling: Can be called without creating any objects using the

syntax : ClassName::FunctionName().

Example: Static Members in Action

```
class Counter {
private:
   static int count; // Declaration
public:
   Counter() { count++; } // Increment shared variable

   static int getCount() { // Static function
      return count; // Can only access static 'count'
   }
};

// Crucial: Outside initialization
int Counter::count = 0;

int main() {
   Counter c1, c2;
   // Call using class name (recommended)
```

```
    std::cout << "Total Objects: " << Counter::getCount(); // Returns 2
}
```

Array of Objects

An Array of Objects allows you to store and manipulate a collection of objects of the same class under a single identifier using contiguous memory.

1. Conceptual Overview

Definition: Just as you can create an array of int or float, you can create an array where each element is an instance of a user-defined class.

Memory Allocation: When an array of objects is declared, memory for all elements is allocated at once.

Constructor Triggering: The default constructor (a constructor with no arguments) is automatically called for every single object in the array at the moment of declaration.

2. Declaration and Initialization

A. Static Array (Stack)

Used when the number of objects is known at compile-time.

Syntax: ClassName arrayName[size];

Example:

```
class Student {
public:
    string name;
    void set(string n) { name = n; }
};
int main() {
    Student classRoom[3]; // Array of 3 objects
    classRoom[0].set("Alice");
    classRoom[1].set("Bob");
    return 0;
}
```

B. Initializing with Parameterized Constructors

You can initialize array elements with specific values using an initialization list.

```
class Point {
   int x, y;
public:
   Point(int xVal, int yVal) : x(xVal), y(yVal) {}
};


// Calling parameterized constructors for each element
Point graph[2] = { Point(1, 2), Point(3, 4) };
```

3. Accessing Elements

You access individual objects using the index operator [] and then use the dot operator . to access that specific object's members.

```
for(int i = 0; i < 3; i++) {
   classRoom[i].display(); // Calls display() for each object
}
```

4. Dynamic Array of Objects (Heap)

For large-scale applications, objects are often created on the heap using the new keyword to prevent stack overflow.

Creation: Employee* staff = new Employee[50];

Deletion: delete[] staff; (The [] is critical to ensure every object's destructor is called).

5. Advanced Study Example: Array of Objects with Static Counter

This example demonstrates how an array of objects interacts with a static member to track a "Global Count."

```
#include <iostream>
using namespace std;
class Sensor {
   int id;
   static int totalSensors; // Shared count
```

```
public:
    Sensor() {
        id = ++totalSensors;
    }
    void status() {
        cout << "Sensor ID " << id << " is ACTIVE." << endl;
    }
};
int Sensor::totalSensors = 0;
int main() {
    // Declaring an array of 5 objects
    // This will trigger the constructor 5 times
    Sensor hub[5];
    cout << "Initializing Sensor Hub..." << endl;
    for(int i = 0; i < 5; i++) {
        hub[i].status();
    }
    return 0;    }
```

6. Critical Implementation Rules

Default Constructor Requirement: If you define any parameterized constructor, you must also define a default constructor (or use default arguments) if you plan to declare a simple array like Class arr[10];. Otherwise, the compiler will throw an error.

Memory Management: When using dynamic arrays (new[]), always use delete[]. Using a simple delete will only destroy the first object, leading to memory leaks and undefined behavior 1.

Efficiency: For massive collections (thousands of objects), consider using std::vector<Class> from the C++ Standard Library, which manages memory growth automatically and is the standard for 2026 development.

Friend Functions

Friend Functions remain a specific architectural tool in C++ used to bypass the standard encapsulation rules. While the principle of "Data Hiding" is central to OOP, certain situations require a non-member function to have full access to a class's private internals.

1. Conceptual Overview

Definition: A friend function is a function that is not a member of a class but has permission to access its private and protected members.

The "Friendship" Grant: Friendship is granted by the class, not taken by the function. To make a function a friend, you must declare it inside the class body using the friend keyword.

Non-Member Status: Even though it is declared inside the class, it is not a member function. It does not have a this pointer and is called like a normal global function.

2. Key Properties of Friend Functions

Scope: It is not in the scope of the class to which it is a friend.

Invocation: It cannot be called using an object (e.g., obj.friendFunc() is invalid). It is called as friendFunc(obj).

Arguments: Usually, it takes an object of the class as an argument to access its members.

Placement: It can be declared in either the public or private section of the class without changing its functionality.

3. Syntax and Example: Basic Friend Function

In this example, we use a friend function to access a private variable for a simple calculation.

```
#include <iostream>
using namespace std;
class Distance {
private:
   int meters;
public:
   Distance() : meters(0) {}
     // Friend function declaration
   friend int addFive(Distance d);
};
```

```
// Definition of friend function (No scope resolution :: used)
int addFive(Distance d) {
   // Accessing private data 'meters' directly
   d.meters += 5;
   return d.meters;
}
int main() {
   Distance dist;
   cout << "Distance after adding 5: " << addFive(dist) << "m"; // Global call
   return 0;
}
```

4. Advanced Use: Bridging Two Different Classes

One of the most common uses for friend functions in 2026 is performing operations that involve two unrelated classes.

```
class ClassB; // Forward declaration
class ClassA {
   int valueA = 10;
   friend void compare(ClassA, ClassB);
};
class ClassB {
   int valueB = 20;
   friend void compare(ClassA, ClassB);
};
// This function is a friend to BOTH classes
void compare(ClassA a, ClassB b) {
   if (a.valueA > b.valueB) cout << "Class A is larger";
   else cout << "Class B is larger";
}
```

5. Pros and Cons for Development

Advantages	Disadvantages
Allows operator overloading for types like cout << obj.	Weakens Encapsulation: Excessive use makes code harder to maintain.
Bridges two classes effectively without complex inheritance.	Non-Member: Does not follow the "Object.Action" syntax of OOP.
Can access private data of multiple classes simultaneously.	Cannot be inherited by subclasses.

6. Important Rules to Remember

Friendship is not mutual: If Class A is a friend of Class B, Class B is not automatically a friend of Class A.

Friendship is not transitive: If A is a friend of B, and B is a friend of C, A is not automatically a friend of C.

Friendship is not inherited: A friend of a base class does not become a friend of the derived class.

Function Overloading

Function Overloading remains a fundamental pillar of C++ development, enabling "Compile-time Polymorphism." It allows you to define multiple functions with the same name but different behaviors based on the data they process.

1. Conceptual Overview

Definition: Overloading occurs when two or more member functions in the same class have the same name but different signatures.

The "Signature": A function signature includes the function name and its parameter list (the number, type, and order of arguments).

Static Binding: The compiler determines which function to call at compile-time by matching the arguments provided in the code to the available signatures.

2. Rules for Overloading

To successfully overload a member function, the functions must differ in at least one of the following:

Number of parameters: (e.g., one int vs. two ints).

Type of parameters: (e.g., an int vs. a double).

Sequence of parameters: (e.g., int, double vs. double, int).

Important Note: Changing the return type alone is not enough to overload a function. The compiler will throw an error because it cannot distinguish which function to call based solely on the return value.

3. Detailed Example: Multi-Purpose Logger

This example shows a class that uses overloading to handle different types of data inputs using a single function name, logData.

```
#include <iostream>

#include <string>

using namespace std;

class DataManager {


public:
   // Overload 1: Handles an integer
   void logData(int id) {
      cout << "Logging Integer ID: " << id << endl;
   }


   // Overload 2: Handles a string (Different Type)
   void logData(string name) {
      cout << "Logging String Name: " << name << endl;
   }


   // Overload 3: Handles two parameters (Different Number)
   void logData(string name, int id) {
      cout << "Logging Pair - Name: " << name << ", ID: " << id << endl;
   }


   // Overload 4: Handles different order (Different Sequence)
```

```
    void logData(int id, string name) {
        cout << "Logging Pair (Reversed) - ID: " << id << ", Name: " << name << endl;
    }
};

int main() {
    DataManager dm;

    dm.logData(101);              // Calls Overload 1
    dm.logData("System_Alpha");   // Calls Overload 2
    dm.logData("Admin", 5);       // Calls Overload 3
    dm.logData(7, "Guest");       // Calls Overload 4

    return 0;
}
```

4. Special Case: Overloading and Default Arguments

You must be careful when combining overloading with default arguments. If the compiler cannot distinguish between an overloaded function and a function with default values, it will result in an ambiguity error.

```
class Demo {
public:
    void display(int x) { ... }
    void display(int x, int y = 10) { ... } // Error-prone
};
// Calling demo.display(5);
// The compiler won't know whether to call the first function
// or the second one with the default 'y' value.
```

5. Why use Function Overloading?

Readability: It prevents the need for clumsy names like printInt(), printString(), and printDouble(). You can simply use print().

Consistency: It provides a unified interface for similar actions performed on different data types.

Maintainability: Code is cleaner and easier to manage as the logic for "printing" or "calculating" is grouped under a single name.

6. Summary Table

Feature	Function Overloading
Binding Type	Compile-time (Static)
Name	Must be identical
Parameters	Must be different (Type, Number, or Order)
Return Type	Can be different, but not the only difference
Scope	Within the same class

Bit Fields

Bit Fields remain an essential C++ feature. They allow developers to specify the exact number of bits used by class data members, significantly reducing the memory footprint of objects.

1. Conceptual Overview

Definition: A bit field is a class data member whose size is specified in bits.

Purpose: To pack multiple variables into a single memory word (like a 4-byte int). This is critical for embedded systems, network headers, and hardware registers where memory is constrained.

Data Types: Traditionally, only integral types (like bool, int, unsigned int, char) can be used as bit fields.

2. Syntax and Declaration

Bit fields are declared by following the member name with a colon (:) and the number of bits.

```
class DeviceStatus {
public:
   // Specify the number of bits after the colon
   unsigned int isReady : 1;    // 1 bit (0 or 1)
   unsigned int mode    : 2;    // 2 bits (0 to 3)
   unsigned int errorID : 4;    // 4 bits (0 to 15)
};
```

3. Detailed Example: System Flags

In this example, we see how bit fields can condense multiple status flags into a single memory block.

```
#include <iostream>
using namespace std;
class Controller {
public:
   // Total bits = 1+1+2+4 = 8 bits (1 byte)
   // Without bit fields, these 4 variables would use 4-16 bytes!
   unsigned int powerOn    : 1;
   unsigned int wifiActive : 1;
   unsigned int speedMode  : 2; // Can store 0, 1, 2, 3
   unsigned int batteryPct : 4; // Can store 0 to 15
   void display() {
      cout << "Power: " << powerOn << " | Wifi: " << wifiActive
         << " | Speed: " << speedMode << " | Battery: " << batteryPct << "/15" << endl;
   }
};
int main() {
   Controller myIoT;
   myIoT.powerOn = 1;
```

```
    myIoT.wifiActive = 0;
    myIoT.speedMode = 3;  // Maximum value for 2 bits
    myIoT.batteryPct = 12;
    myIoT.display();
       cout << "Size of object: " << sizeof(myIoT) << " byte(s)" << endl;
    return 0;
}
```

4. Important Constraints and Rules

Addressing: You cannot take the address of a bit field member using the & operator because they might not start at a byte boundary.

Array Limit: You cannot create an array of bit fields.

Over-assignment: Assigning a value larger than the bit capacity results in truncation or undefined behavior. (e.g., assigning 5 to a 2-bit field results in 1).

Alignment: The compiler may add padding bits between fields to ensure alignment with the machine's word boundaries.

5. Memory Comparison: Bit Fields vs. Regular Members

If we define a class with four unsigned int members normally:

Regular Class:

4×4 bytes=16 bytes (128 bits)4 cross 4 bytes equals 16 bytes (128 bits)

4×4 bytes=16 bytes (128 bits)

Bit Field Class: If the fields total 8 bits, the compiler allocates only

1 byte (8 bits)1 byte (8 bits)

1 byte (8 bits)

or

4 bytes4 bytes

4 bytes

(depending on the underlying type and padding).

6. Special Case: Unnamed Bit Fields

You can use unnamed bit fields for padding to align subsequent members to specific bit positions.

```
class Header {
   unsigned int type : 4;
   unsigned int      : 4; // Unnamed padding to fill the byte
   unsigned int code : 8;
};
```

7. When to use ?

Embedded Drivers: Writing to specific bits of a hardware port.

Memory-Mapped I/O: Interfacing with sensors.

Network Protocols: Constructing packet headers (e.g., IPv4/IPv6 flags) efficiently according to RFC standards.

Constructors/Destructors

Constructors/Destructors and Static Members remains the standard pattern for "Object Tracking" and "Resource Management." While constructors initialize individual instances, static members maintain the global state of the entire class.

1. The Core Concept

Constructor's Role: Executed every time a new object is created. In this pattern, it is used to increment a shared counter or initialize shared resources.

Destructor's Role: Executed every time an object goes out of scope or is deleted. It is used to decrement the shared counter or release shared resources once the last object is destroyed.

Static Member: Acts as the "Global Registry" that persists across all instances.

2. Implementation: Object Tracker Example

This is the most common use case: keeping track of how many instances of a class currently exist in memory.

```
#include <iostream>
using namespace std;
class Widget {
private:
   static int activeCount; // 1. Declare static variable
   int id;
public:
```

```
    // CONSTRUCTOR
    Widget() {
        id = ++activeCount; // 2. Increment shared counter
        cout << "Widget " << id << " created. Total: " << activeCount << endl;
    }
    // DESTRUCTOR
    ~Widget() {
        --activeCount;      // 3. Decrement shared counter
        cout << "Widget " << id << " destroyed. Remaining: " << activeCount << endl;
    }
    static int getCount() { return activeCount; }
};
// 4. CRITICAL: Initialize static member outside the class
int Widget::activeCount = 0;
int main() {
    cout << "--- Global Scope ---" << endl;
    Widget w1, w2;
    {
        cout << "--- Local Scope Start ---" << endl;
        Widget w3;
        cout << "Active: " << Widget::getCount() << endl;
    } // w3 is destroyed here automatically

    cout << "--- Local Scope End ---" << endl;
    cout << "Final Active: " << Widget::getCount() << endl;
    return 0;
}
```

3. Summary Table

Event	Action on Static Member	Result
Object Instantiation	Increment in Constructor	Tracks total "living" objects.
Object Destruction	Decrement in Destructor	Maintains accurate real-time count.
Static Function Call	Read-only access	Provides class-wide status without an object.

UNIT - III

Operator Overloading

Operator Overloading remains a powerful tool in C++ for making user-defined types (classes/structs) behave like built-in types. It allows you to redefine the behavior of existing operators for your objects.

1. Unary vs. Binary Operator Overloading

Operators are categorized by their arity (number of operands they accept), which cannot be changed during overloading.

Unary Operators: These operate on a single operand (e.g., ++, --, -, !).

Member Function: Takes zero explicit arguments (the calling object is implicit).

Friend Function: Takes one explicit argument.

Binary Operators: These operate on two operands (e.g., +, -, *, /, ==).

Member Function: Takes one explicit argument (the second operand); the first operand is the calling object.

Friend Function: Takes two explicit arguments.

Example: Unary and Binary Overloading

```
class Vector {
    int x, y;
public:
    Vector(int x=0, int y=0) : x(x), y(y) {}
    // Unary Overload (Member): Negation (-)
    Vector operator-() { return Vector(-x, -y); }
    // Binary Overload (Member): Addition (+)
    Vector operator+(const Vector& v) { return Vector(x + v.x, y + v.y); }
};
```

2. Overloading Using Friend Functions

Friend functions are non-member functions that have access to a class's private data. They are highly recommended for binary operators to ensure symmetry (e.g., allowing 5 + complex as well as complex + 5).

Rule: Friend functions must be declared inside the class with the friend keyword but defined outside (or as an inline friend).

Arity difference: Because there is no implicit this pointer, friend functions require one extra parameter compared to member functions.

Example: Binary Addition as Friend

```
class Complex {
   double real, imag;
public:
   Complex(double r, double i) : real(r), imag(i) {}
   friend Complex operator+(const Complex& c1, const Complex& c2);
};
Complex operator+(const Complex& c1, const Complex& c2) {
   return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

3. Type Conversion

C++ continues to support three primary levels of data conversion involving classes:

Basic Type to Class Type: Handled by a single-argument constructor.

Example: MyClass obj = 10; (Calls MyClass(int)). Use the explicit keyword to prevent accidental conversions.

Class Type to Basic Type: Handled by a Conversion Operator.

Syntax: operator type() { return value; }

Example: Converting a Fraction class to a double.

Class Type to Class Type: Handled either by a constructor in the destination class or a conversion operator in the source class.

Example: Class to Basic Type

```
class Weight {
   double kg;
public:
   Weight(double k) : kg(k) {}
```

```
    // Conversion Operator: Weight to double
    operator double() const { return kg; }
};
Weight w(5.5);
double d = w; // Automatic conversion to 5.5
```

Restrictions and Rules

Non-Overloadable: ., .*, ::, ?:, and sizeof cannot be overloaded.

Must be Member: =, [], (), and -> can only be overloaded as member functions, not as friends.

Precedence: You cannot change the original precedence or associativity of operators

Inheritance

Inheritance remains a pillar of C++ Object-Oriented Programming, allowing a new class (derived class) to acquire the properties and behaviors of an existing class (base class). This promotes code reusability and establishes a logical hierarchy in complex systems.

Types of Inheritance in C++

1. Single Inheritance

A single child class inherits from exactly one parent class. It is the most direct form of inheritance.

Example: A Dog class inheriting from an Animal class.

Code Structure: class Dog : public Animal { ... };.

2. Multiple Inheritance

A single child class inherits from more than one base class simultaneously. This allows the child to combine features from multiple sources.

Example: A SmartPhone class inheriting from both a Camera class and a Phone class.

Code Structure: class SmartPhone : public Camera, public Phone { ... };.

3. Multilevel Inheritance

A class is derived from another derived class, forming a chain or lineage.

Example: Grandparent → Parent → Child.

Code Structure: class B : public A { ... }; followed by class C : public B { ... };.

4. Hierarchical Inheritance

Multiple child classes inherit from the same single parent class. This is used when different objects share common core attributes.

Example: Both Car and Bus inherit from the Vehicle class.

Code Structure: class Car : public Vehicle { ... }; and class Bus : public Vehicle { ... };.

5. Hybrid Inheritance

A combination of two or more of the above inheritance types (e.g., combining Multilevel and Multiple inheritance).

Example: A FlyingCar might use multiple inheritance from Car and FlyingMachine, where Car already inherits from Vehicle.

Code Structure: class D : public B, public C { ... }; where B and C might have their own separate base classes.

6. Multi-path Inheritance (The Diamond Problem)

A specific form of hybrid inheritance where a child class inherits from two parents that share a common grandparent. This creates ambiguity because the child inherits two separate copies of the grandparent's members.

The Diamond Solution: In 2026, the standard way to resolve this is using the virtual keyword during inheritance to ensure only one shared copy of the grandparent exists.

Syntax: class B : virtual public A { ... }; and class C : virtual public A { ... };.

Summary of Inheritance Properties

Inheritance Type	Structure	Key Use Case
Single	1 Base → 1 Derived	Simple "is-a" relationships.
Multiple	Many Base → 1 Derived	Combining distinct functionalities.
Multilevel	Chain (A → B → C)	Step-by-step specialization.
Hierarchical	1 Base → Many Derived	Sharing common behaviors.
Hybrid	Mixed types	Modeling complex real-world roles.
Multi-path	Diamond shape	Advanced hierarchies requiring virtual inheritance.

Below is a detailed breakdown of each inheritance type with specific code examples.

1. Single Inheritance

A single derived class inherits from exactly one base class. This is the most basic form of inheritance.

Example: A Car inherits features from a Vehicle.

```
class Vehicle {
public:
   void engine() { cout << "Engine started" << endl; }
};

class Car : public Vehicle { // Inherits only from Vehicle
public:
   void brand() { cout << "Ford Mustang" << endl; }
};
```

2. Multiple Inheritance

A single derived class inherits from two or more base classes simultaneously. This allows a class to combine distinct sets of behaviors.

Example: A SmartWatch inherits features from both Watch and DigitalDevice.

```
class Watch {
public:
   void showTime() { cout << "12:00 PM" << endl; }
};
class DigitalDevice {
public:
   void connectWifi() { cout << "Connected" << endl; }
};
class SmartWatch : public Watch, public DigitalDevice { // Multiple parents
};
```

3. Multilevel Inheritance

A derived class inherits from another derived class, forming a chain or lineage.

Example: Animal → Mammal → Human.

```
class Animal {
public:
   void eat() { cout << "Eating..." << endl; }
};
class Mammal : public Animal { // Intermediate class
public:
   void breathe() { cout << "Breathing..." << endl; }
};
class Human : public Mammal { // Inherits from both Mammal and Animal
};
```

4. Hierarchical Inheritance

Multiple derived classes inherit from the same single base class. This is useful when different objects share core attributes but have unique behaviors.

Example: Car and Bus both inherit from Vehicle.

```
class Vehicle {
public:
   void fuel() { cout << "Consuming fuel" << endl; }
};
class Car : public Vehicle { }; // First child
class Bus : public Vehicle { }; // Second child
```

5. Hybrid Inheritance

A combination of two or more inheritance types, such as joining multiple and multilevel inheritance within the same structure.

Example: Combining Single and Multiple inheritance.

```
class A { };
```

class B : public A { }; // Single Inheritance

class C { };

class D : public B, public C { }; // Hybrid (B+C Multiple)

6. Multi-path Inheritance (The Diamond Problem)

A specific hybrid structure where a class inherits from two parents that both share a common grandparent. This causes ambiguity because the child gets two separate copies of the grandparent's data.

Solution: Use the virtual keyword during inheritance to ensure only one shared copy of the base class exists.

class GrandParent { public: int x; };

class Parent1 : virtual public GrandParent { }; // Virtual inheritance

class Parent2 : virtual public GrandParent { }; // Virtual inheritance

class Child : public Parent1, public Parent2 { }; // Ambiguity resolved

Virtual Base Classes and Abstract Classes are essential for designing robust, scalable C++ architectures. They solve two distinct problems: memory ambiguity in complex hierarchies and the enforcement of architectural contracts.

Virtual Base Classes

Virtual Base Classes are used in Multi-path Inheritance to prevent the "Diamond Problem."

The Problem: The Diamond Problem

If class B and class C both inherit from class A, and class D inherits from both B and C, class D will contain two separate copies of class A. This causes ambiguity (which copy should be used?) and wastes memory.

The Solution: Virtual Inheritance

By declaring the base class as virtual, C++ ensures that only one instance of the base class is shared by all classes in the inheritance path.

Example:

#include <iostream>

using namespace std;

class PoweredDevice {

public:

```
    int powerLevel = 100;
};
// Use 'virtual' to ensure shared instance
class Scanner : virtual public PoweredDevice {};
class Printer : virtual public PoweredDevice {};
class Copier : public Scanner, public Printer {
public:
    void checkPower() {
        // Without 'virtual', this would cause a compiler error (ambiguity)
        cout << "Power level: " << powerLevel << endl;
    }
};
int main() {
    Copier myOfficeMachine;
    myOfficeMachine.checkPower();
    return 0;
}
```

Abstract Classes

An Abstract Class is a class designed specifically to be a base class. It cannot be instantiated (you cannot create an object of this class).

Key Characteristics

Pure Virtual Function: A class becomes abstract if it contains at least one pure virtual function.

Syntax: virtual void functionName() = 0;

Purpose: It acts as a blueprint or "Interface." It defines what a class should do, but leaves the how to the derived classes.

Enforcement: Any class inheriting from an abstract class must override all pure virtual functions, or it will also become an abstract class.

Example: Shape Interface

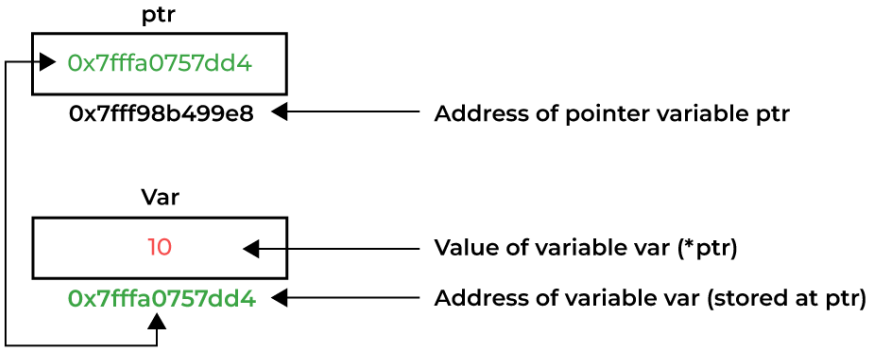
```
#include <iostream>
```

```
using namespace std;
// ABSTRACT CLASS
class Shape {
public:
   // Pure Virtual Function
   virtual void draw() = 0;
     // Abstract classes can still have regular functions
   void info() { cout << "This is a shape." << endl; }
};
class Circle : public Shape {
public:
   void draw() override { // Implementation required
      cout << "Drawing a Circle..." << endl;
   }
};
class Square : public Shape {
public:
   void draw() override {
      cout << "Drawing a Square..." << endl;
   }
};
int main() {
   // Shape s; // ERROR: Cannot instantiate abstract class
      Shape* s1 = new Circle();
   Shape* s2 = new Square();
   s1->draw(); // Polymorphic call
   s2->draw();
      delete s1; delete s2;
   return 0;
}
```

UNIT - IV

Pointers

Pointers remain a fundamental feature of C++ and C, essential for low-level memory manipulation, efficient data handling, and dynamic memory allocation.



Core Definition

Pointer Variable: A special variable that stores the memory address of another variable instead of a direct value.

Memory Address: A unique hexadecimal identifier (e.g., 0x7ffee4b0a98c) representing a specific location in the computer's RAM where a variable's value is stored.

Key Operators

Address-of Operator (&): Prefixed to a variable name to retrieve its memory address (e.g., &var).

Indirection/Dereference Operator (*):

In Declaration: Used to designate a variable as a pointer (e.g., int *ptr).

In Usage: Used to access or modify the actual value stored at the address held by the pointer (e.g., *ptr = 20).

Pointer Declaration and Initialization

A pointer must be declared with a data type that matches the type of the variable it points to.

Declaration: int *ptr; (ptr is a pointer to an integer).

Initialization: ptr = &var; (stores the address of var in ptr).

Single-line Definition: int *ptr = &var;.

Pointer Arithmetic

Pointers support limited arithmetic operations, which move the pointer by the size of the data type it points to (not just by 1 byte).

Increment (ptr++): Moves the pointer to the address of the next element of that type.

Subtraction (p1 - p2): Finds the number of elements between two pointers of the same type.

Special Pointer Types

NULL/nullptr Pointer: A pointer that points to no valid memory location (address 0). It is best practice to initialize unused pointers to nullptr to prevent crashes.

Void Pointer (void*): A "generic" pointer that can hold the address of any data type but must be typecast before dereferencing.

Dangling Pointer: A pointer that points to a memory location that has been deallocated or freed, leading to undefined behavior.

Wild Pointer: An uninitialized pointer that holds a random, unpredictable memory address.

Declaration Of Pointers

The declaration of pointers remains the gateway to manual memory management in C++. A pointer declaration tells the compiler three things: the name of the pointer, that it is a pointer, and the type of data it is allowed to point to.

1. Basic Syntax of Pointer Declaration

To declare a pointer, use the asterisk (*) symbol between the data type and the pointer name.

Syntax: data_type *pointer_name;

data_type: The type of variable the pointer will point to (e.g., int, char, double).

*: The pointer operator that tells the compiler this variable stores an address.

pointer_name: The identifier for the pointer.

2. Examples of Declaration

A. Integer and Float Pointers

```
int *iPtr;    // Pointer to an integer
float *fPtr;  // Pointer to a float
double *dPtr; // Pointer to a double
```

B. Initialization during Declaration

In 2026, it is a strict best practice to initialize pointers immediately. A pointer declared without an address is a "wild pointer" and can crash your program.

int age = 25;

int *agePtr = &age; // Declared and initialized to the address of 'age'

C. The nullptr (The Safe Way)

If you don't have an address to assign yet, initialize the pointer to nullptr. This ensures the pointer doesn't point to a random, dangerous memory location.

int *p = nullptr; // Declared as a null pointer (Address 0)

3. Key Rules for Pointer Declaration

Type Matching: A pointer must match the type of the variable it points to. You cannot point an int* to a double variable without explicit casting.

double salary = 5000.50;

// int *ptr = &salary; // ERROR: Cannot convert double* to int*

double *ptr = &salary; // Correct

The Position of the Asterisk: In C++, the asterisk can be placed in three ways, and they all mean the same thing to the compiler:

int* ptr; (Preferred by many 2026 developers as it emphasizes the type is "int-pointer")

int *ptr; (Commonly used in C-style)

int * ptr;

Declaring Multiple Pointers: Be careful when declaring multiple pointers on a single line. The * applies only to the variable immediately following it.

int *p1, p2; // p1 is a pointer, p2 is a regular integer

int *p3, *p4; // p3 and p4 are both pointers

4. Advanced Declarations (2026 Context)

Pointer to Pointer: A variable that stores the address of another pointer.

int val = 10;

int *ptr = &val;

int **ptrToPtr = &ptr; // Double pointer

Constant Pointers:

```
const int *ptr;      // The data pointed to cannot be changed
int *const ptr = &x;  // The address stored in the pointer cannot be changed
```

Understanding how pointers interact with class hierarchies is essential for mastering Runtime Polymorphism. These concepts allow C++ to decide which function to execute at the moment the program runs.

The this Pointer

The this pointer is an implicit parameter passed to all non-static member functions. It is a constant pointer that holds the memory address of the current object invoking the function.

Implicit Usage: When you access a member variable inside a class, the compiler secretly uses this->variableName.

Disambiguation: Its most common use is distinguishing between class members and local parameters when they have the same name.

Chaining: It can be used to return the current object (e.g., return *this;) to allow method chaining.

Example:

```
class Player {
   int score;
public:
   void setScore(int score) {
      // 'this->score' refers to the class member
      // 'score' refers to the function parameter
      this->score = score;
   }
};
```

2. Pointers to Base and Derived Classes

C++ allows a specific relationship between pointers of classes in an inheritance hierarchy.

A. Base Class Pointer to Derived Object (Upcasting)

A pointer of a Base class type can point to an object of a Derived class. This is the foundation of polymorphism.

Limitation: By default, the pointer can only access members defined in the Base class. It cannot see "extra" members added by the Derived class unless virtual functions are used.

B. Derived Class Pointer to Base Object (Downcasting)

Typically, you cannot point a Derived class pointer to a Base class object. This is because a Base object lacks the extra features a Derived pointer expects. This requires an explicit dynamic_cast to be safe.

3. Virtual Functions and Late Binding

To make a Base class pointer actually execute the Derived class's version of a function, the function must be declared virtual in the Base class.

Comprehensive Example:

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void show() { // Virtual keyword enables late binding
        cout << "Displaying Base class" << endl;
    }
};
class Derived : public Base {
public:
    void show() override { // 'override' ensures correct signature in 2026
        cout << "Displaying Derived class" << endl;
    }
};
int main() {
    Base* ptr;        // Base pointer
    Derived obj;      // Derived object
    ptr = &obj;       // Pointing Base pointer to Derived object
```

```
    // Because show() is virtual, the Derived version is called
    ptr->show();       // Output: "Displaying Derived class"
    return 0;
}
```

Summary for Development

Concept	Key Feature	Best Practice
this Pointer	Points to the active object.	Use to resolve naming conflicts in constructors.
Base Pointer	Can point to any Derived class.	Use to create collections of different objects (e.g., std::vector<Base*>).
Virtual Function	Enables Runtime Polymorphism.	Always use the override keyword in Derived classes for safety.
Virtual Destructor	Ensures proper cleanup.	Always define a virtual ~Base() = default; if your class has virtual functions.

Array

In C++, an array is a fundamental data structure used to store multiple elements of the same data type in a contiguous block of memory. Instead of declaring separate variables for each value, arrays allow you to manage a collection under a single name.

Characteristics of Arrays

Homogeneous Elements: All elements must be of the same data type (e.g., all int, all double, or all objects of the same class).

Fixed Size: The size of a static array must be defined at compile-time and cannot be changed during program execution.

Contiguous Memory: Elements are stored in adjacent memory locations, which enables random access—reaching any element in constant time

Zero-Based Indexing: The first element is always at index 0, and the last element is at size - 1.

No Built-in Bounds Checking: C++ does not verify if an index is within range; accessing arr[10] in a 5-element array results in undefined behavior.

Examples of Array Declaration and Initialization

```
// Static declaration with size 5
int marks[5];
// Declaration with immediate initialization
int numbers[5] = {10, 20, 30, 40, 50};
// Partial initialization (rest are set to 0)
int partial[5] = {1, 2}; // {1, 2, 0, 0, 0}
```

Characteristics of Arrays:

Arrays remain a fundamental building block in C++ for managing collections of data efficiently. The core characteristics that define how they operate in memory and within code are detailed below.

1. Homogeneous Data Type

An array can only store elements of the same data type. You cannot mix different types (e.g., an int and a char) within a single array.

Example: int marks[5]; can only store integers.

2. Contiguous Memory Allocation

Elements are stored in adjacent (consecutive) memory locations. This layout is "cache-friendly" and allows the system to calculate the exact address of any element quickly.

Memory View: If an int takes 4 bytes and arr[0] is at address 1000, arr[1] will be at 1004, arr[2] at 1008, and so on.

3. Constant Time Access (Random Access)

Because of contiguous memory, any element can be reached in O(1) time. You do not need to traverse the entire list to find the middle element; you simply use its index.

Example: cout << arr[500]; takes the same time as cout << arr[0];.

4. Fixed Size (Static Allocation)

For standard arrays, the size must be known at compile-time and cannot be changed during program execution. If you need a flexible size, you must use dynamic allocation with the new operator or use a std::vector.

Example: Once int arr[10]; is declared, it will always hold exactly 10 slots until the program or scope ends.

5. Zero-Based Indexing

C++ uses 0-based indexing, meaning the first element is at index 0 and the last element is at index size - 1.

Example: In int arr[5], valid indices are 0, 1, 2, 3, 4.

6. No Built-in Bounds Checking

C++ does not check if an index is within the valid range. Accessing arr[10] in a 5-element array results in undefined behavior, which may cause crashes or return "garbage" values from unrelated memory.

Verification: Modern developers often use the std::vector::at() method to enforce bounds checking.

7. Pointer Compatibility (Array Decay)

The name of an array acts as a constant pointer to its first element. When passed to a function, the array "decays" into a pointer, losing its size information.

Example: int* ptr = arr; is valid and makes ptr point to arr[0].

8. Multi-Dimensional Support

Arrays can be "nested" to create tables (2D) or cubes (3D) of data.

2D Example: int matrix[3][4]; creates 3 rows and 4 columns.

Characteristic	Key Behavior
Indexing	Starts at 0, ends at n−1n minus 1 $n-1$.
Performance	O(1) access for any element.
Memory	All elements sit side-by-side in RAM.
Modification	Can update elements via index: arr[2] = 10;.

Array Of Objects

An array of objects remains a primary method for grouping multiple instances of a class under a single identifier in C++. Each element in such an array is a distinct object of the same class type, stored in contiguous memory locations.

Key Characteristics

Contiguous Storage: Objects are placed side-by-side in memory. For example, in an array Obj arr[10], if the size of Obj is 32 bytes, each subsequent object is exactly 32 bytes further in memory.

Automatic Construction: When you declare a static array of objects, C++ automatically calls the default constructor for every element in that array. If no default constructor exists and you don't provide initializers, the code will fail to compile.

Individual State: While the array shares a name, each object maintains its own unique set of data members.

Fixed Size: Like standard arrays, the size of a static array of objects must be known at compile-time and cannot be changed later.

Declaration and Initialization Examples

1. Static Declaration (Default Constructor)

If your class has a default constructor, declaration is simple.

cpp

```
class Employee {
public:
   int id;
   Employee() : id(0) {} // Default Constructor
};
Employee staff[50]; // Declares 50 objects, each initialized to id 0
```

2. Using Parameterized Constructors (Initializer List)

You can initialize specific objects with different values at the time of declaration.

```
class Box {
public:
   int length;
```

```
    Box(int l) : length(l) {}
};
// Initializing 3 objects with specific lengths
Box warehouse[3] = {Box(10), Box(20), Box(30)};
```

3. Dynamic Allocation

For cases where the number of objects is only known at runtime, use the new operator.

```
int size;
std::cin >> size;
Employee* workers = new Employee[size]; // Allocated on heap
// Usage
workers[0].id = 101;
delete[] workers; // MUST use delete[] to free memory
```

Accessing Member Functions

Individual objects in the array are accessed using their index, followed by the dot operator.

```
for(int i = 0; i < 3; i++) {
    warehouse[i].calculateVolume(); // Calls method for each object
}
```

Comparison: Advantages vs. Disadvantages

Feature	Benefit/Drawback
Random Access	Efficient O(1) access to any object via its index.
Memory Efficiency	Reduced overhead compared to storing individual pointers for each object.
Fixed Size	Cannot grow if more objects are needed; requires re-allocation.
Constructor Overhead	Every object is constructed immediately upon declaration, which can be slow for very large arrays.

Memory Model

In C++, the term memory model generally refers to two distinct concepts: the physical memory layout of a program and the multithreaded memory consistency model introduced in C++11.

1. Program Memory Layout (Physical Model)

When a C++ program is loaded into RAM, the Operating System assigns it a block of memory divided into specific segments:

Text (Code) Segment: Stores the compiled read-only machine instructions. This prevents accidental modification of the program's logic during execution.

Data Segment: Holds global and static variables. It is further split into:

Initialized Data: For variables with explicit starting values (e.g., int x = 10;).

BSS (Uninitialized Data): For variables initialized to zero or not initialized at all in the code.

Stack: A LIFO (Last-In, First-Out) structure used for local variables, function parameters, and return addresses. It is managed automatically by the compiler; memory is reclaimed as soon as a function finishes.

Heap: A large pool used for dynamic memory allocation (via new or malloc). The programmer is responsible for manual deallocation using delete or free to prevent memory leaks.

2. C++11 Multithreaded Memory Model

This is a formal "contract" between the programmer and the system that defines how memory operations (reads and writes) are observed across different threads.

Core Guarantees

Atomic Operations: Standardized via <atomic>, these ensure specific operations occur as a single, indivisible step, preventing data races.

Modification Order: Every atomic object has a defined order of changes that all threads will eventually agree upon.

Synchronizes-With: A relationship where a "write" in one thread is guaranteed to be visible to a "read" in another, establishing a "happens-before" link.

Standard Memory Orderings

C++ provides different levels of strictness for these operations:

Sequential Consistency (memory_order_seq_cst): The default and strictest. All threads see all operations in the same global order.

Acquire/Release (memory_order_acquire / memory_order_release): A middle ground. It ensures that all memory writes performed by thread A before it releases a lock are visible to thread B once it acquires that same lock.

Relaxed (memory_order_relaxed): The weakest. Only guarantees that the operation itself is atomic; it does not impose any ordering on other surrounding memory operations.

Operators for Dynamic Memory

In C++, the new and delete operators are used for dynamic memory management. They allow you to request memory from the heap (free store) at runtime rather than at compile-time, which is essential when the exact amount of memory needed is unknown beforehand.

1. The new Operator

The new operator performs two main tasks: it allocates a sufficient block of memory on the heap and, for classes, automatically calls the constructor to initialize the object.

Syntax & Basic Usage:

Single Variable: int* p = new int; (Allocates space for one integer).

With Initialization: int* p = new int(21); (Allocates space and sets the value to 21).

Arrays: int* arr = new int[10]; (Allocates memory for a block of 10 integers).

Error Handling: By default, if the heap is exhausted and allocation fails, new throws a std::bad_alloc exception.

Alternative: Using new (std::nothrow) will return a nullptr instead of throwing an exception if it fails.

2. The delete Operator

The delete operator releases the memory previously allocated by new back to the system. For objects, it calls the destructor before freeing the memory.

Syntax & Basic Usage:

Single Variable: delete p;.

Arrays: delete[] arr; (Note the square brackets; these are mandatory when deleting arrays to ensure all element destructors are called).

Best Practice: After deleting, set the pointer to nullptr to avoid "dangling pointers" (pointers that still point to freed memory).

3. Key Comparisons

Feature	new / delete	malloc() / free() (C-style)
Type Safety	Returns a pointer of the exact type.	Returns a void* that requires casting.
Object Lifecycle	Calls constructors and destructors.	Does not call constructors/destructors.
Syntax	Operator based (standard C++).	Function based (requires <cstdlib>).
Failure	Throws an exception by default.	Returns NULL.

4. Advanced Usage: Placement New

Placement new is a variation that allows you to construct an object at a pre-allocated memory address instead of allocating new space on the heap.

Syntax: MyClass* obj = new (preallocated_ptr) MyClass();

Use Case: Critical in low-level systems programming or custom memory pools where performance is vital.

Common Errors to Avoid

Memory Leaks: Forgetting to use delete leads to memory staying reserved even after the program no longer needs it.

Mismatching: Never use free() on memory allocated with new, or delete on memory from malloc().

Double Deletion: Attempting to delete the same pointer twice causes undefined behavior and likely crashes.

For modern applications in 2026, manual use of new and delete is often discouraged in favor of smart pointers (like std::unique_ptr or std::shared_ptr) which manage memory automatically

Dynamic Object

A dynamic object in C++ is an instance of a class or structure that is created at runtime using dynamic memory allocation. Unlike static objects, which are allocated on the stack with a fixed

lifetime, dynamic objects are allocated on the heap and persist until they are explicitly destroyed by the programmer.

Core Characteristics

Heap Allocation: Dynamic objects reside in the heap memory segment rather than the stack.

Controlled Lifetime: Their existence is not limited by the scope (like a function's { } brackets) in which they were created; they remain in memory until delete is called.

Pointer Access: Because they are created at runtime, they are always accessed via pointers that store their memory address.

Manual Deallocation: The compiler does not automatically reclaim memory for dynamic objects. You must use the delete operator to avoid memory leaks.

Creating and Using Dynamic Objects

1. Basic Declaration

Use the new operator followed by the class name. This automatically triggers the class constructor to initialize the object.

// Syntax: ClassName ptr = new ClassName();*

Employee* dev = new Employee(); // Dynamically creates an Employee object

2. Parameterized Initialization

Dynamic objects can be initialized with specific values at runtime.

// Calls the parameterized constructor

Box* package = new Box(10, 20, 30);

3. Accessing Members

Because you are working with a pointer, you must use the arrow operator (->) to access the object's methods or data.

package->calculateVolume(); // Accessing a member function

4. Deallocation

Use delete to call the destructor and free the heap memory.

delete package;

package = nullptr; // Good practice to avoid dangling pointers

In C++, the concepts of binding, polymorphism, and virtual functions work together to allow objects to behave differently depending on their actual type at runtime.

1. Binding

Binding is the process of connecting a function call to the specific function definition that should be executed.

Static Binding (Early Binding): The compiler links the function call to its definition at compile-time based on the type of the pointer or object. This is the default in C++ and is used for non-virtual functions.

Example: If you call a regular function obj.show(), the compiler knows exactly which show() to call before the program even runs.

Dynamic Binding (Late Binding): The function call is resolved at runtime based on the actual object the pointer is pointing to. This is required for runtime polymorphism and is achieved using virtual functions.

2. Polymorphism

Polymorphism means "many forms". It allows a single interface (like a function name) to handle different underlying data types or behaviors.

Compile-time Polymorphism: Achieved via Function Overloading (same name, different parameters) or Operator Overloading (giving new meaning to operators like +).

Runtime Polymorphism: Achieved via Function Overriding with Virtual Functions. It allows a base class pointer to call a derived class's version of a function.

Example of Polymorphism:

```
void print(int i) { cout << "Integer: " << i; }  // Function Overloading

void print(string s) { cout << "String: " << s; } // (Compile-time)
```

3. Virtual Functions

A virtual function is a member function in a base class that you expect to be redefined (overridden) in derived classes. When called through a base class pointer, C++ uses the actual object's type to decide which version to run.

VTable and VPtr: Behind the scenes, the compiler creates a VTable (Virtual Table) for each class with virtual functions, containing pointers to those functions. Every object of that class has a hidden VPtr (Virtual Pointer) that points to its class's VTable to find the correct function at runtime.

Detailed Code Example:

```
class Animal {

public:
```

```
    virtual void sound() { // Virtual function
        cout << "Some generic animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override { // Overriding base function
        cout << "Woof! Woof!" << endl;
    }
};

int main() {
    Animal* myPet = new Dog();
    myPet->sound(); // Output: "Woof! Woof!" due to Dynamic Binding
    delete myPet;
    return 0;
}
```

UNIT - V

File Handling in C++

File handling means reading from and writing to files (like .txt, .csv, etc.) using classes provided by the C++ standard library.

Programs run in RAM, meaning data exists only while the program is running, when a program ends, all data in RAM is lost automatically.

File handling allows storing data in secondary memory (like HDD or SSD), so it can be preserved permanently and can be saved and accessed even after the program terminates.

For file operations, C++ provides file stream classes in the <fstream> header such as ofstream, ifstream, fstream.

Opening a File

Before reading from or writing to a file, we first need to open it. Opening a file loads that file in the RAM. In C++, we open a file by creating a stream to it using the fstream class that represent the file stream i.e. stream for input and output to the file.

fstream str("filename.ext", mode); where, str: Name given to the stream

filename: Name of the file

mode: Represents the way in which we are going to interact with the file.

File Opening Modes

File opening mode indicate file is opened for reading, writing, or appending. Below is the list of all file modes in C++:

Mode	Description
ios::in	File open for reading. If file does not exists, the open operation fails.
ios::out	File open for writing: the internal stream buffer supports output operations.
ios::binary	Operations are performed in binary mode rather than text.
ios::ate	The output position starts at the end of the file.
ios::app	All output operations happen at the end of the file, appending to its existing contents.
ios::trunc	Any contents that existed in the file before it is open are discarded.

For Example, if we want to open the file for reading, we use the following opening mode:

fstream filein("file.txt", ios::in);

Similarly, if we want to open the file for writing, we use the following:

fstream fileout("file.txt", ios::out);

These modes can also be combined using OR operator (|). For example, you can open the file stream in both read and write mode as shown:

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

If the file opened in write mode does not exists, a new file is created. But if the file opened in read mode doesn't exists, then no new file is created, and an exception is thrown

Other File Streams

fstream is not the only file stream provided by C++. There are two more specialized streams:

ifstream: Stands for input file stream. It is equivalent to open fstream in ios::in mode.

ofstream: Stands for output file stream. It is equivalent to opening fstream in ios::out mode.

The above modes are default modes for these streams. These modes cannot be changed but can be clubbed together with other modes. Now for input, we can also use ifstream as shown:

ifstream filein("file.txt");

Similarly, for output:

ofstream fileout("file.txt");

Write Data to File

Once the file is opened in the write mode using either fstream or ofstream, we can perform the write operation in similar way as with cout using << operator.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
  // Open a file
  ofstream file("GFG.txt");
    // Write the string to the file
  file << "Welcome to KWC.";
  return 0;
```

```
}
```

Read Data from File

Once the file is opened in the read mode using either fstream or ifstream, we can perform the write operation in similar way as with cin using >> operator.

```
#include <bits/stdc++.h>

using namespace std;

int main()

{

   // Open a file in read mode

   ifstream file("GFG.txt");

   string s;

   // Read string from the file

   file >> s;

   cout << "Read String: " << s;

   return 0;

}
```

Output

```
Read String: Welcome
```

Closing the File

Closing the file means closing the associated stream and free the resources that we being used. It is important to close the file after you are done with it, especially in the long running programs to avoid memory leaks, data loss, etc.

In C++, the files are closed using the close() member function that is present in all file streams.

```
#include <bits/stdc++.h>

using namespace std;

int main()

{

   // Open a file in read mode

   ifstream file("GFG.txt");
```

```
    string s;
    // Read string from the file
    getline(file, s);
    cout << "Read String: " << s;
    // Close the file
    file.close();
    return 0;
}
```

Output

Read String: Welcome to KWC.

Errors in File Handling

Many different types of errors can occur in file handling such as file not found, disk full, etc. Our programs should expect common errors and should be able to handle them properly. Following are some common errors that can occur during file handling:

File Open Failure

There can be cases in which the file is not opened due to various reasons such as it doesn't exists, or the program does not have permission to open it, etc. In this case, we can use the is_open() member function of the file stream classes to check whether the file is opened sucessfullly or not.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    fstream file("nonexistent_file.txt", ios::in);
    // Check if the file is opened
    if (!file.is_open()) {
        cerr << "Error: Unable to open file!" << endl;
        return 1;
    }
    file.close();
```

```
    return 0;
}
```

Output

Error: Unable to open file!

Failure to Read/Write Data

Another common error is failure to read or write data for reasons such as incorrect mode, etc. In this case, we can validate operations after each read/write attempt. For example, reading using getline() can be validated as shows:

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    fstream file("GFG.txt", ios::out);
    if (!file.is_open()) {
        cerr << "Error: Unable to open file!" << endl;
        return 1;
    }
    string line;
        // Checking if getline() read successfully or not
    if (!getline(file, line))
        cerr << "Error: Failed to read data" << endl;
    file.close();
    return 0;
}
```

Output

Error: Failed to read data

End-of-File (EOF) Error

Trying to read beyond the end of the file can cause an EOF error. This can happen when you don't check for the end of the file before reading. We can check for EOF using eof() member function.

```
#include <bits/stdc++.h>
```

```
using namespace std;

int main()
{
    ifstream file("GFG.txt");
    if (!file.is_open())
    {
        cerr << "Error: Unable to open file!" << endl;
        return 1;
    }
    string line;
    while (getline(file, line))
        cout << line << endl;
    // Check for eof
    if (file.eof())
        cout << "Reached end of file." << endl;
    else
        cerr << "Error: File reading failed!" << endl;
    file.close();
    return 0;
}
```

Output

Reached end of file.

Notice that we have also validated the read operation before checking EOF as getline() will only return nullptr even if the read fails due to any reason.

Handling Binary Files

In C++, we can also handle binary files, which store data in raw format. To read and write binary data, must use the ios::binary flag when creating/opening a binary file.

Write into Binary File

To write data to a binary file, we first need to open or create the file in ios::binary mode.

```
#include <cstring>
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   string str = "Welcome to GeeksForGeeks";
   // Open a binary file for writing
   ofstream file("fileBin.bin", ios::binary);
   // Check if the file is open
   if (!file)
   {
      cerr << "Error opening the file for writing.";
      return 1;
   }
   // Write the length of the string (size) to file first
   size_t strLength = str.length();
   file.write(reinterpret_cast<const char *>(&strLength), sizeof(strLength));
   // Write the string to the binary file
   file.write(str.c_str(), strLength);
   // Close the file
   file.close();
   return 0;
}
```

Output

Reading from Binary File

Just as we open a file in binary mode to write data, to read data from a binary file, we must open the file in read mode using ios::in.

Syntax:

fstream fileInstance("fileName.bin", ios::in| ios::binary);

```
#include <cstring>
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   string str;
   // Open the binary file for reading
   fstream file("fileBin.bin", ios::in | ios::binary);
   // Check if the file is open
   if (!file)
   {
      cerr << "Error opening the file for reading.";
      return 1;
   }
   // Read the length of the string (size) from the file
```

```
    size_t strLength;
    file.read(reinterpret_cast<char *>(&strLength), sizeof(strLength));
    // Allocate memory for the string and read the data
    char *buffer = new char[strLength + 1]; // +1 for the null-terminator
    file.read(buffer, strLength);
    // Null-terminate the string
    buffer[strLength] = '\0';
    // Convert the buffer to a string
    str = buffer;
    // Print file data
    cout << "File Data: " << str;
    delete[] buffer;
    file.close();
    return 0;
}
```

Output

File Data: Welcome to KWC

Standard Template Library (STL) in C++

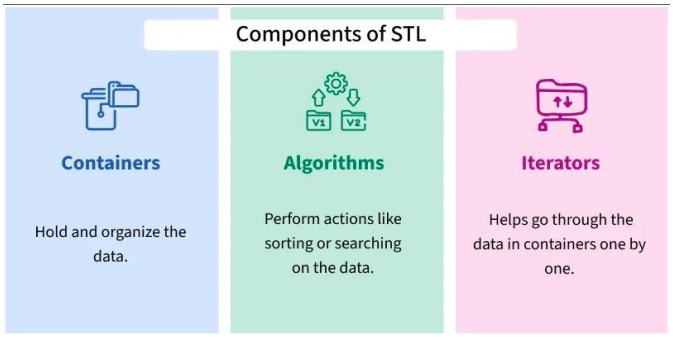
STL is a collection of pre-built classes and functions that make it easy to manage data using common data structures like vectors, stacks, and maps. It saves time and effort by providing ready-to-use, efficient algorithms and containers.

Components of STL

The components of STL are the features provided by STL in C++ that can be classified into 3 types:

These components are designed to be efficient, flexible, and reusable, making them an integral part of modern C++ programming.

Containers



Containers are the data structures used to store objects and data according to the requirement. Each container is implemented as a template class that also contains the methods to perform basic operations on it. Every STL container is defined inside its own header file.

Containers can be further classified into 4 types:

- **Sequence Containers :** Vector, Deque, List, Forward List, Array
- **Container Adaptors :** Stack, Queue, Priority Queue
- **Associative Containers :** Set, Multiset, Map, Multimap
- **Unordered Associated Containers :** Unordered Set, Unordered Multiset, Unordered Map, Unordered Multimap

Instead of writing your own data structures (like linked lists or stacks), STL provides ready-made containers that are: Fast, Reliable, Easy-to use and Type-Safe (work with any data type using templates).

Algorithms

STL algorithms offer a wide range of functions to perform common operations on data (mainly containers). These functions implement the most efficient version of the algorithm for tasks such as sorting, searching, modifying and manipulating data in containers, etc. All STL algorithms are defined inside the <algorithm> and <numeric> header file. Some of the most frequently used algorithms are:

Sort : Arranges elements in ascending order (default).

Binary Search : Checks whether a value exists in a sorted range.

Find : Searches for the first occurrence of a given value.

Count : Counts how many times a value appears in the given range.

Reverse : Reverses the order of elements in the given range.

Accumulate : Computes the sum of all elements in the range.

Unique : Removes consecutive duplicate elements.

Lower bound : Returns iterator to the first element ≥ value in a sorted range.

Upper bound : Returns iterator to the first element > value in a sorted range.

Replace : Replaces all occurrences of old value with new value in the given range.

Iterators

Iterators are the pointer like objects that are used to point to the memory addresses of STL containers. They are one of the most important components that contributes the most in

connecting the STL algorithms with the containers. Iterators are defined inside the <iterator> header file.

Benefits of C++ Standard Template Library (STL)

The key benefits of the STL is:

- Saves time and effort.
- Reliable and Tested
- Fast and Efficient
- Reusability
- Built-in Algorithms

Strings in C++

Strings in C++ are objects of the std::string class. They are used to represent and manipulate sequences of characters.

Unlike C-style character arrays (char[]), std::string handles memory management automatically and provides a wide range of built-in functions for ease of use.

Can automatically grow and shrink as you add or remove characters, unlike fixed-size C-style strings.

You can easily access characters, join strings, compare them, extract substrings, and search within strings using built-in functions.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    // Creating a string
  string str = "Hello Folks";
  // Traversing using index
  cout << "Using index: ";
  for (int i = 0; i < str.size(); i++) {
    cout << str[i] ;
  }
```

```
    cout << endl;
    // Traversing using range-based for loop
    cout << "Using range-based for loop: ";
    for (char ch : str) {
        cout << ch ;
    }
    cout << endl;

    // Traversing using iterator
    cout << "Using iterator: ";
    for (auto it = str.begin(); it != str.end(); it++) {
        cout << *it ;
    }
    cout << endl;
    return 0;
}
```

Output

Using index: Hello Folks

Using range-based for loop: Hello Folks

Using iterator: Hello Folks

Syntax

The string container is defined as std::string class inside the <string> header file.

string str;

where,

string: Class provided by STL to handle sequences of characters.

str: Name assigned to the string object.

Basic Operations in String

Basic operations on Strings are shown below:

Initializing a String

Initialization of a string assigns characters to the string at the time of creation.

A string can be initialized directly using = or constructor syntax with text inside quotes.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
      // Initializing a string directly
   string str = "Hello Folks";
   // Printing the string
   cout << str << endl;
   return 0;
}
```

Output

Hello Folks

Accessing Characters

Characters of a string can be accessed using the [] operator or the .at() function.

Time complexity for accessing characters is O(1).

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string str = "Hello Geeks";
   // Access using index operator []
   cout << "First character: " << str[0] << endl;
   cout << "Fifth character: " << str[4] << endl;
   // Access using at()
   cout << "Character at index 6: " << str.at(6) << endl;
```

```
    return 0;
}
```

Output

```
First character: H
Fifth character: o
Character at index 6: F
```

String Length

The number of characters in a string can be found using size() or length().

Time complexity to find string length is O(1).

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str = "Hello Geeks";
    // Using size()
    cout << "Length using size(): " << str.size() << endl;
    // Using length()
    cout << "Length using length(): " << str.length() << endl;
    return 0;
}
```

Concatenation of Strings

Strings can be joined using the + operator or the append() function.

The + operator creates a new string, while append() modifies the existing string in place.

Time complexity for concatenation is O(n+m), where n is the size of string and m is the size of the string to be concatenated.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string str1 = "Hello";
   string str2 = " Folks";
   // Using + operator
   string result1 = str1 + str2;
   cout << "Concatenation using + : " << result1 << endl;
   // Using append() function
   string result2 = str1;
   result2.append(str2);
   cout << "Concatenation using append(): " << result2 << endl;
   return 0;
}
```

Output

```
Concatenation using + : Hello Folks
Concatenation using append(): Hello Folks
```

Modifying a String

Characters of a string can be added with .push_back(), removed with .pop_back(), or altered using .insert() and .erase().

Time complexity for push/pop is O(1) and O(n) for insert/erase.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string str = "Hello Folks";
```

```
    // Adding a character at the end
    str.push_back('!');
    cout << "After push_back: " << str << endl;
    // Removing the last character
    str.pop_back();
    cout << "After pop_back: " << str << endl;
    // Inserting a substring
    str.insert(5, " C++");
    cout << "After insert: " << str << endl;
    // Erasing part of the string
    str.erase(5, 4);
    cout << "After erase: " << str << endl;
    return 0;
}
```

Output

After push_back: Hello Folks!

After pop_back: Hello Folks

After insert: Hello C++ Folks

After erase: Hello Folks

Substring Extraction

The .substr(pos,len) is used to extract a part of a string, where pos means the starting position and len means how many characters you want to copy.

This function creates a new string containing the selected portion, starting at pos and copying len characters.

Time complexity of extraction is O(len).

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

```
int main() {
    string str = "Hello Folks";
    // Extract "Hello"
    string sub1 = str.substr(0, 5);
    cout << "Substring 1: " << sub1 << endl;
    // Extract "Folks"
    string sub2 = str.substr(6, 5);
    cout << "Substring 2: " << sub2 << endl;
    return 0;
}
```

Output

```
Substring 1: Hello
Substring 2: Folks
```

Searching in a String

The find() function is used to search for a substring inside a string. If found, it returns the index (position) where the substring starts; if not, it returns a special value (npos).

Time complexity to search is O(n*m), where n is the length of string and m is the substring length.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str = "Hello Folks";
    size_t pos = str.find("Folks");
    if (pos < str.size()) {
        cout << "\"Folks\" found at index: " << pos << endl;
    }
    return 0;
}
```

Output

"Folks" found at index: 6

C++ String Functions

C++ provides some inbuilt functions which are used for string manipulation, such as the strcpy() and strcat() functions for copying and concatenating strings. Some of them are:

Function	Description
length()	This function returns the length of the string.
swap()	This function is used to swap the values of 2 strings.
size()	Used to find the size of string
resize()	This function is used to resize the length of the string up to the given number of characters.
find()	Used to find the string which is passed in parameters
push_back()	This function is used to push the passed character at the end of the string
pop_back()	This function is used to pop the last character from the string
clear()	This function is used to remove all the elements of the string.
strncmp()	This function compares at most the first num bytes of both passed strings.
strncpy()	This function is similar to strcpy() function, except that at most n bytes of src are copied
strrchr()	This function locates the last occurrence of a character in the string.
strcat()	This function appends a copy of the source string to the end of the destination string
find()	This function is used to search for a certain substring inside a string and returns the position of the first character of the substring.
replace()	This function is used to replace each element in the range [first, last) that is equal to old value with new value.
substr()	This function is used to create a substring from a given string.
compare()	This function is used to compare two strings and returns the result in the form of an integer.
erase()	This function is used to remove a certain part of a string.
rfind()	This function is used to find the string's last occurrence.