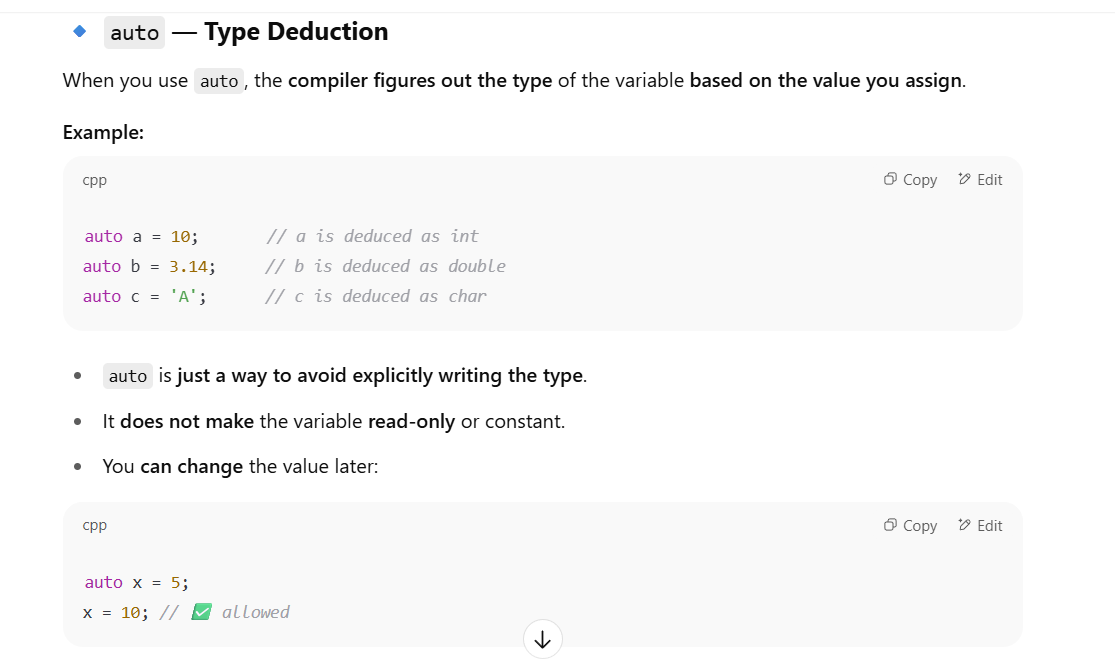
# CPP

🡺C++ is a **general-purpose programming language** developed by **Bjarne Stroustrup** in 1979 as an extension of C. It adds **Object-Oriented Programming (OOP)** to C.

🡺C++ is known as a **multi-paradigm** language because it supports different ways of programming.

🡺**1998** – First ISO Standard (C++98)

🡺auto z = 10; // C++ automatically infers z is an int



Namespaces solve the problem of multiple libraries or parts of a program using the same name for different entities, by organizing them into distinct scopes.

**List initialization (int x{10};)**

* **Explanation:** List initialization (brace initialization) introduced in C++11 is preferred because it's uniform and, more importantly, prevents implicit narrowing conversions that can lead to data loss.

|| is the logical OR operator. & and | are bitwise operators, and && is the logical AND operator.

🡺When an array is initialized with fewer elements than its declared size, the remaining elements are automatically initialized to zero (for numeric types).

🡺 most built-in non-numeric types when an array is partially initialized, the remaining elements are still initialized to a "zero-equivalent" value (like \0 for char, false for bool, nullptr for pointers). For user-defined types, it generally involves calling their default constructor or zero-initialization.

Accessing an array element using an index that is outside its declared bounds (e.g., arr[10] for int arr[5];) in C++ results in:

a) A compile-time error.

b) A runtime error that always crashes the program.

c) Undefined behavior. 🡺

d) Automatic resizing of the array.v explain

Which statement is true about new operator?

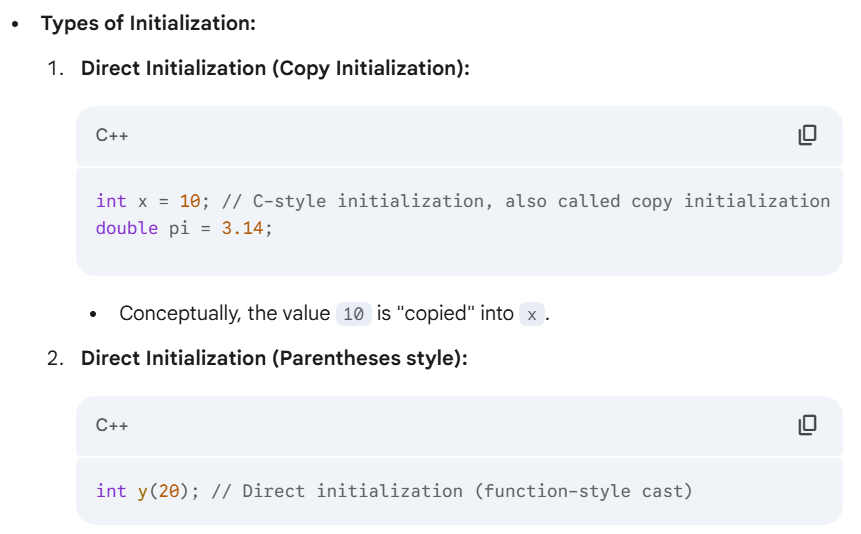
A. Returns void\*

B. Returns exception on failure 🡺

C. Allocates memory on stack

D. Needs typecastin

By default, if the new operator fails to allocate the requested memory (e.g., due to memory exhaustion), it throws a std::bad\_alloc exception. You can use a try-catch block to handle this. (There's also a new (std::nothrow) version that returns nullptr on failure instead of throwing, but the standard new throws an exception).



3 🡺List Initialization (Uniform Initialization / Brace Initialization - C++11 onwards):

int z{30}; // Preferred for consistency and preventing narrowing conversions

double e{}; // Value initialization (initializes to zero for built-in types)

This is the **preferred modern C++ way**. It's more uniform and prevents **narrowing conversions** (e.g., trying to put a double into an int without explicit cast, which would lose precision). If a narrowing conversion is attempted with brace initialization, the compiler will issue an error.

**Zero-Initialization / Value-Initialization:**

* + For global, static, and thread-local variables, if not explicitly initialized, they are automatically **zero-initialized** (set to 0 or equivalent).
  + For local (automatic) variables, if not initialized, they contain **garbage values**.
  + Using {} for initialization (e.g., int value{};) explicitly value-initializes the variable to 0 (for numeric types), false (for bool), or nullptr (for pointers).

**Importance:** Always initialize your variables to avoid undefined behavior.

# CPP Notes

🔹1. **The Need for C++**

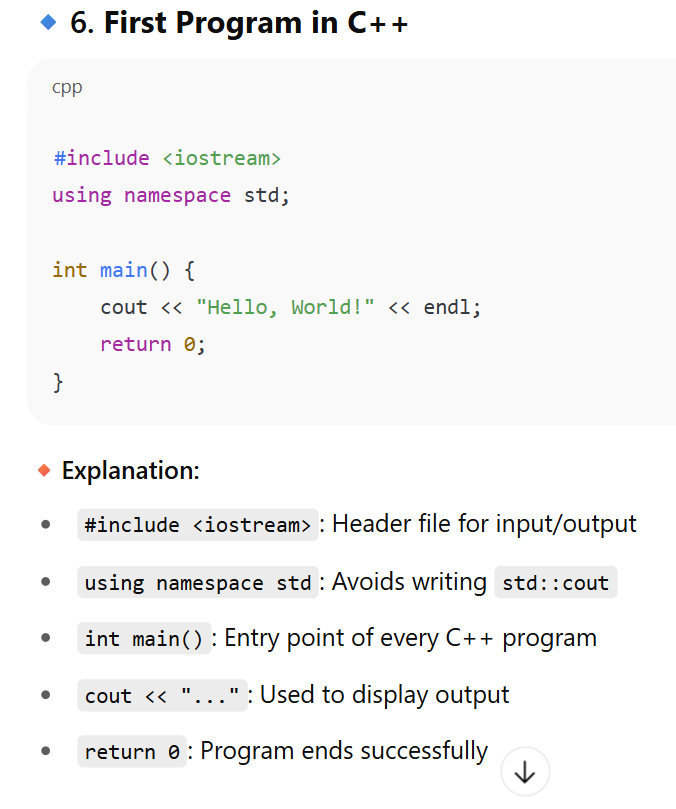
C++ is a **general-purpose programming language** developed by **Bjarne Stroustrup** in 1979 as an extension of C. It adds **Object-Oriented Programming (OOP)** to C.

C++ is known as a **multi-paradigm** language because it supports different ways of programming.

| **Feature** | **Description** |
| --- | --- |
| OOP | Supports classes, objects, inheritance, polymorphism |
| Speed | Very fast, close to machine-level |
| Compiled | Compiled using tools like GCC or Visual Studio |
| Portable | Runs across different platforms (Windows, Linux) |
| STL | Has **Standard Template Library** (we’ll learn later) |
| Memory Management | Uses new and delete operators for dynamic memory |

**🔹4. History of C++**

* **1979** – Bjarne Stroustrup starts developing C with Classes.
* **1983** – Named C++
* **1998** – First ISO Standard (C++98)
* **2011 / 14 / 17 / 20** – Modern C++ updates (C++17 is your syllabus)



**🔍 Components Breakdown:**

| **Part** | **Description** |
| --- | --- |
| #include <iostream> | Preprocessor directive for I/O |
| using namespace std; | Avoids writing std:: again and again |
| int main() | Main function where program starts |
| { ... } | Block of code |
| int a = 10; | Variable declaration |
| cout << ... | Output |
| return 0; | End of program |

**✅ 3. C++ Tokens**

A **token** is the smallest individual unit in a program.

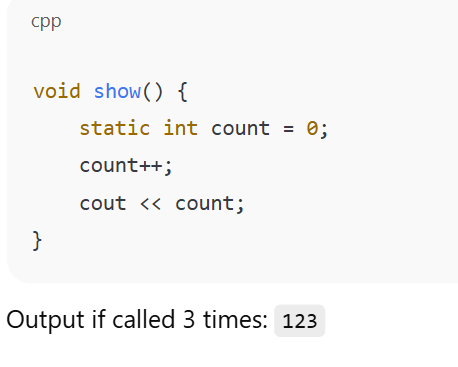
**Types of Tokens:**

1. **Keywords** – Reserved words (e.g., int, return, if)
2. **Identifiers** – Names of variables, functions, etc.
3. **Literals** – Constants (e.g., 10, 3.14, 'A')
4. **Operators** – +, -, \*, /, etc.
5. **Separators** – ;, {}, ()

**✅ 5. Static & Constant Members**

**🔹 static variables**

* Belong to the **class**, not objects
* **Retain value between function calls**



**const keyword**

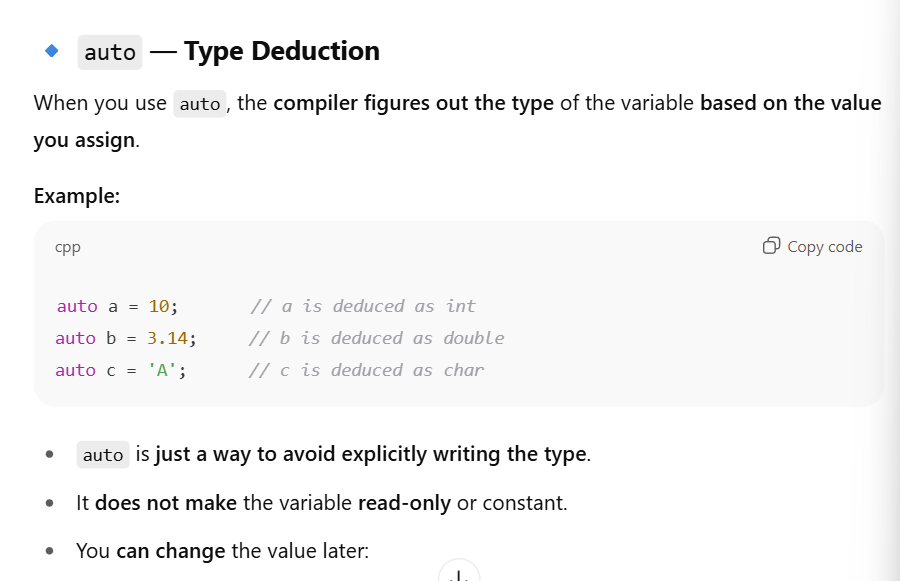
* Defines **constants**, value cannot be changed

**✅ 6. C++ Operators**

Let’s cover all the key types.

| **Type** | **Example** | **Purpose** |
| --- | --- | --- |
| Arithmetic | +, -, \*, /, % | Math |
| Relational | ==, !=, <, >, <=, >= | Comparison |
| Logical | &&, || |  |
| Unary | ++, --, -x, +x | One operand |
| Ternary | cond ? x : y | If-else in one line |
| Assignment | =, +=, -=, \*= | Assign values |

Bitwise op 🡺 |



# 🔹 Session 4: Conditional and Looping Statements

**switch Statement:**

Used for multiple value-based cases:

int ch = 2;

switch (ch) {

case 1: cout << "One"; break;

case 2: cout << "Two"; break;

default: cout << "Invalid";

}

🔹 break is important to stop fall-through.

| **Concept** | **Example** | **Notes** |
| --- | --- | --- |
| if | if (a > b) | Executes if true |
| switch | switch(val) | Use break |
| for | for(int i=0;i<n;i++) | Definite loops |
| while | while(cond) | Conditional loop |
| do-while | do {} while(cond) | Runs once even if false |
| break | Stops loop |  |
| continue | Skips iteration |  |
| 1D Array | int a[5] | Access: a[0] |
| 2D Array | int a[2][3] | Access: a[i][j] |

**What Are Command-Line Arguments?**

They are **values passed to your program from the terminal/command line**, **when you run the program**.

This allows you to give input **without using cin**.



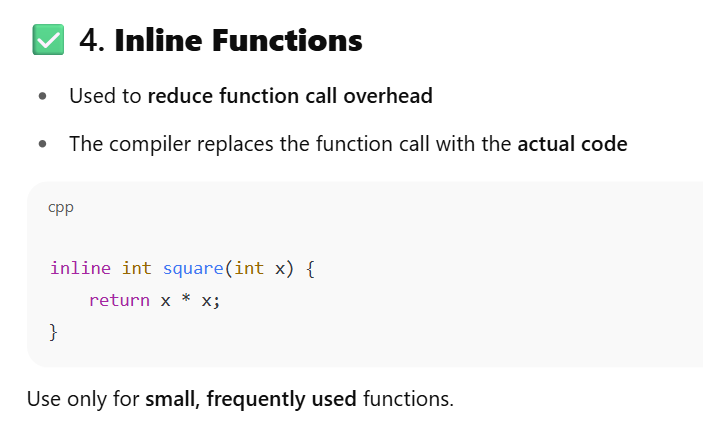
# 🔹 Session 5: Functions in C++

**✅ 1. What is a Function?**

A function is a **block of code** that performs a specific task.

**🔹 Types:**

| **Type** | **Example** |
| --- | --- |
| Built-in | sqrt(), pow(), abs() |
| User-defined | Functions you write yourself |



**✅ 5. Math Library Functions**

| **Function** | **Description** | |
| --- | --- | --- |
| sqrt(x) | Square root | |
| pow(x, y) | x raised to the power y | |
| abs(x) | Absolute value | |
| floor(x) | Rounds down | |
| ceil(x) | Rounds up | |
| fmod(a, b) | Remainder (like % but for float) | |
| **Concept** | | **Example** | | **Key Point** |
| Declaration | | int add(int, int); | | Tells compiler |
| Definition | | int add(int a, int b) | | Code logic |
| Call | | add(5, 6); | | Executes function |
| Call by Value | | void f(int x) | | Doesn’t affect caller |
| Call by Reference | | void f(int &x) | | Affects caller |
| Inline | | inline int sq(int x) | | Code is inserted in place |

# Memory Management and Pointers

**✅ 1. What is a Pointer?**

A **pointer** is a variable that stores the **address** of another variable.

int x = 10;

int\* ptr = &x;

cout << \*ptr; // Outputs: 10

Symbol Meaning

\* Used to declare and dereference a pointer

& Used to get the address of a variable

**✅ 2. Memory Layout in C++**

| **Segment** | **Stores** |
| --- | --- |
| Stack | Local variables, function calls |
| Heap | Dynamically allocated memory (new, malloc) |
| Code | Instructions |
| Static | Global/static variables |

**✅ 3. Dynamic Memory Allocation using new**

int\* p = new int; // allocates memory for 1 int

\*p = 50;

**🔹 Allocate array using new:**

int\* arr = new int[5]; // Allocates array of 5 integers

arr[0] = 10;

**✅ Deallocation using delete:**

delete p; // Frees memory of single variable

delete[] arr; // Frees memory of array

**✅ 4. Comparison: new vs malloc**

| **Feature** | **new** | **malloc** |
| --- | --- | --- |
| Returns | Exact data type | void\* (needs typecast) |
| Throws | Exception on failure | Returns NULL |
| Syntax | int\* p = new int; | int\* p = (int\*) malloc(sizeof(int)); |
| Free | delete p; | free(p); |

**✅ 5. Pointers and Arrays**

int arr[] = {10, 20, 30};

int\* p = arr;

cout << \*p; // 10

cout << \*(p + 1); // 20

✅ You can treat pointers like array names

**✅ 7. this Pointer**

Used inside a class to refer to **current object**.

class Demo {

int a;

public:

void setA(int a) {

this->a = a; // disambiguates member vs local var

}

};

**✅ 8. Pointer to Class**

class A {

public:

void show() {

cout << "Hello";

}

};

A obj;

A\* ptr = &obj;

ptr->show(); // Accessing using pointer

| **Concept** | **Example** | **Notes** |
| --- | --- | --- |
| Pointer | int\* p = &x; | Holds address |
| Dereference | \*p | Gets value at address |
| Array Pointer | arr, \*(arr+1) | Pointer arithmetic |
| new | int\* p = new int; | Allocates memory |
| delete | delete p; | Deallocates |
| this | this->x | Refers to current object |
| No <string.h> | Manually write string functions using pointers |  |

==================================🡺Gemi🡸===============================

# Introduction to Memory Management in C++

**Static/Global Segment (.data and .bss):**

* **.data segment:** Stores initialized global variables, static variables, and string literals.
* **.bss segment:** Stores uninitialized global variables and static variables (initialized to zero by the system).
* These variables have **static storage duration**, meaning they exist for the entire lifetime of the program.

**Stack Segment:**

* Used for **automatic (local) variables** (declared inside functions), function parameters, and return addresses for function calls.

**Heap (Free Store) Segment:**

* Used for **dynamic memory allocation**. Memory is explicitly requested by the programmer during runtime using new (in C++) or malloc (in C).
* This memory persists until it is explicitly released using delete (in C++) or free (in C) or until the program terminates.

## Pointers in C++

A **pointer** is a variable that stores the memory address of another variable. Instead of holding a value directly, it "points" to where a value is stored.

**Declaration:** dataType\* pointerName;

* The \* indicates that pointerName is a pointer to a variable of dataType.
* **Example:** int\* ptr; declares ptr as a pointer that can hold the address of an int.

Initialize to nullptr (C++11 onwards, preferred) or NULL (C-style macro) to indicate it points to nothing.

int\* ptr = nullptr; // Modern C++

int\* oldPtr = NULL; // C-style, still works

**Address-of Operator (&):**

* Used to get the memory address of a variable.
* **Example:** int num = 10; int\* ptr = &num; (ptr now holds the address of num).

**Dereference Operator (\*):**

* Also called the **indirection operator**. Used to access the value stored at the memory address pointed to by a pointer.
* **Example:** std::cout << \*ptr; (This would print 10).

🡺Cannot add two pointers or multiply/divide pointers.

🡺An uninitialized pointer in C++ is commonly referred to as a(n):" (Options: Null pointer, Valid pointer, **Wild pointer**, Smart pointer)

### Accessing Array Elements via Pointers:

* arr[i] is equivalent to **\*(arr + i)**

### Arrays using Pointers

int arr[] = {10, 20, 30};

int\* p = arr; // p now points to arr[0] (address of 10)

int arr[] = {10, 20, 30};

int\* p = arr;

* + Array accesss Code

std::cout << \*p << std::endl; // Prints 10 (value at arr[0])

std::cout << \*(p + 1) << std::endl; // Prints 20 (value at arr[1])

std::cout << p[2] << std::endl; // Prints 30 (using array-style access with pointer)

p++; // p now points to arr[1]

std::cout << \*p << std::endl; // Prints 20

🡺When an array name arr is used in most expressions, what does it typically **decay** into?" (Options: The first element's value, A pointer to the last element, **A pointer to the first** element, The size of the array)

### **4. Enumeration (enum):**

An enumeration (enum) is a user-defined data type that consists of a set of named integer constants. It makes code more readable and self-documenting than using raw integer values.

By default, the first enumerator is assigned 0, the second 1

Implicit conversion from enum to int is allowed

### Scoped Enums (enum class - C++11 onwards):

* To prevent implicit conversions to int and to avoid name collisions in the global scope, C++11 introduced enum class (scoped enums).

enum class Color { RED, GREEN, BLUE };

// Color c = RED; // Error: RED is not in this scope

Color c = Color::RED; // Correct

// int x = Color::GREEN; // Error: No implicit conversion

🡺This is generally preferred for stronger type safety.

**Mcq**

What is the primary purpose of an enum in C++?"

(Options: To create dynamic arrays,

**To define a set of named integer constants,**

To perform complex calculations,

To manage memory)

The enum class (scoped enum) in C++11 helps to prevent what?" (Options: Runtime errors, Explicit type conversions, **Implicit conversions to int and name collisions**, Compile-time errors)

### Typedef

* **Explanation :**
  + The typedef keyword is used to create an **alias** (a new name) for an existing **data type.** It does not create a new type; it just provides an alternative name for one that already exists.
  + **Syntax:** typedef existing\_type new\_name;
* C++
* typedef int integer;
* integer num = 10; // 'integer' is now an alias for 'int'

**using alias declaration instead of Typedef**

**Syntax: using new\_name = existing\_type;**

**Example: using integer = int;**

### Using new operator

The new operator is used for **dynamic memory allocation** on the heap (free store) in C++. It allocates memory and, for objects, also calls the appropriate constructor.

**Purpose:** To allocate memory whose size is not known at compile time or whose lifetime needs to extend beyond the scope of the function where it's created.

**Syntax:** pointer\_variable = new Type; (for single object) pointer\_variable = new Type[size]; (for array of objects)

**Return Value:** new returns a pointer to the newly allocated memory. If memory allocation fails (out of memory), new by default throws a **std::bad\_alloc exception**. (It can also return nullptr if (std::nothrow) is used).

new Type[size]{};: Value-initializes all elements of the array to zero.

When new is used to allocate memory for an object of a class, what additional operation does it perform besides allocating raw memory?" (Options: Calls the destructor, **Calls the constructor**, Calls a random function, Performs garbage collection)

### Class pointer

Just like you can have pointers to built-in types (like int\*), you can also have pointers to objects (instances) of classes.

**Declaration:** ClassName\* objectPointer;

 **Creating Object and Pointing:**

1. **Stack-allocated object:**

C++

MyClass obj; // Object created on the stack

MyClass\* ptr = &obj; // Pointer points to stack object

1. **Heap-allocated object:**

C++

MyClass\* ptr = new MyClass(); // Object created on the heap via new

**Accessing Members through Class Pointers:**

* You cannot use the dot (.) operator with a pointer to an object.
* You must use the **arrow operator (->)** to access members (data or functions) of the object pointed to by a class pointer.
* Alternatively, you can dereference the pointer first and then use the dot operator: (\*objectPointer).member (but -> is much more common and readable).

### 8. this pointer

It's a **const pointer**: You cannot change this to point to a different object.

It's **implicit**: You don't declare it; it's automatically provided by the compiler.

**Not available in static member functions**: Because static member functions belong to the class, not a specific object, they don't have an associated this pointer.

### 9. Comparison of new over malloc, calloc and realloc, etc.

* C++ supports both its own new/delete operators and the C-style memory management functions (malloc, calloc, realloc, free). While both allocate memory, new/delete are strongly preferred in C++ due to their object-oriented features.

**malloc (memory allocate):**

* Allocates a block of memory of a specified size (in bytes).
* Does NOT call constructors.
* malloc: Allocates memory but does not initialize its contents; the memory contains "garbage" (whatever was previously there).

**calloc (contiguous allocate):**

* Allocates a block of memory for a specified number of elements of a certain size, and **initializes all bytes to zero**.
* **calloc**: Allocates memory for a specified number of elements and *initializes all bytes in the allocated memory block to zero*.
*  Returns void\*, or NULL on failure.
*  **Does NOT call constructors.**

**realloc (re-allocate):**

Changes the size of a previously allocated memory block. It can expand or shrink it.

Does NOT call constructors/destructors for relocated objects.

realloc: Changes the size of a previously allocated memory block. It does not guarantee zero-initialization for the entire block, only for newly extended portions if the new size is larger than the old.

**free:**

* Deallocates memory previously allocated by malloc, calloc, or realloc.
* **Does NOT call destructors.**
* **Example:** free(arr);

free: Deallocates previously allocated memory.

### 10. Memory freeing using Delete operator

*  The delete operator is the counterpart to new. It is used to **deallocate (free)** memory that was previously allocated on the heap using new.
*  **Crucial:** For every new, there must be a corresponding delete. For every new[] (array allocation), there must be a corresponding delete[]. Mismatched new/delete or new[]/delete leads to **undefined behavior**.
*  **Purpose:** To prevent **memory leaks** (where allocated memory is no longer accessible but remains occupied, reducing available memory).

**🡺This prevents dangling pointers (pointers that point to deallocated memory)**

After using delete on a pointer, what is a good practice to prevent dangling pointers?" (Options: Reassign it to another valid address, **Set the pointer to nullptr**, Increment the pointer, Decrement the pointer)

🡺 Attempting to delete the same memory address twice can lead to:" (Options: Increased performance, A memory leak, A safe no-operation**, Undefined behavior**)

#### Quick Recap: Sessions 6 & 7

* **Memory Segments:** Program memory is divided into Code, Static/Global, Stack (for automatic variables, LIFO, limited size), and Heap (for dynamic allocation, flexible size, manual management).
* **Pointers:** Variables storing memory addresses. Declared with \*, address obtained with &, value accessed with \* (dereference). Support pointer arithmetic based on data type size.
* **Arrays & Pointers:** Array names often decay to pointers to their first element. arr[i] is equivalent to \*(arr + i). Arrays of pointers hold addresses.
* **Enumerations (enum):** User-defined types with named integer constants, improving readability. enum class for stronger type safety (C++11).
* **typedef:** Creates type aliases for readability and portability. using (C++11) is a modern alternative.
* **new Operator:** Allocates dynamic memory on the heap. Returns typed pointer. Calls constructors for objects. Throws std::bad\_alloc on failure by default. Used for single objects (new Type;) or arrays (new Type[size];).
* **Class Pointers:** Pointers to objects. Members are accessed using the -> (arrow) operator (e.g., ptr->member).
* **this Pointer:** Implicit, constant pointer in non-static member functions, pointing to the current object. Used to distinguish members from local variables and return the current object. Not available in static member functions.
* **new/delete vs. malloc/free:** new/delete are preferred in C++ for type safety, automatic constructor/destructor calls, operator overloading, and exception handling. malloc/free are C-style, lack these C++-specific benefits.
* **delete Operator:** Deallocates memory allocated by new. Must be paired correctly (delete for new, delete[] for new[]). Essential to prevent memory leaks. Good practice: set pointer to nullptr after delete to prevent dangling pointers and double-free issues.

# Session 8 - Classes and Objects

### 1. Introduction to Classes and Objects

**Class:**

* A **class** is a blueprint or a template for creating objects. It's a user-defined data type.
* A class itself does not occupy memory; it's just a definition.

**Object:**

* An **object** is an instance of a class. It's a concrete realization of the blueprint.
* When an object is created, memory is allocated for its data members.

**Access Specifiers:** These keywords (public, private, protected) control the visibility and accessibility of class members.

**Class Declaration Syntax:** "Which of the following is the correct syntax for declaring a class named Book in C++?" (class Book { };)

**Default Access:** "What is the default access specifier for members of a class in C++ if none is explicitly specified?" (Options: public, private, protected, none)

**private**

* For class declarations in C++, if you don't explicitly specify an access specifier (like public, private, or protected) for the first members, they are private by default.
* (Note: For struct declarations, the default access specifier is public.)

### Defining Data Members and Member Functions

**Data Members:**

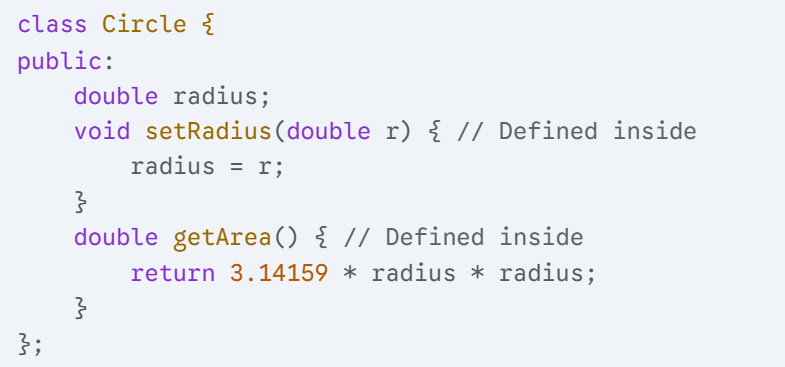
* Variables declared within a class are its data members. They represent the **attributes or properties of the objects** created from that class.

**Member Functions:**

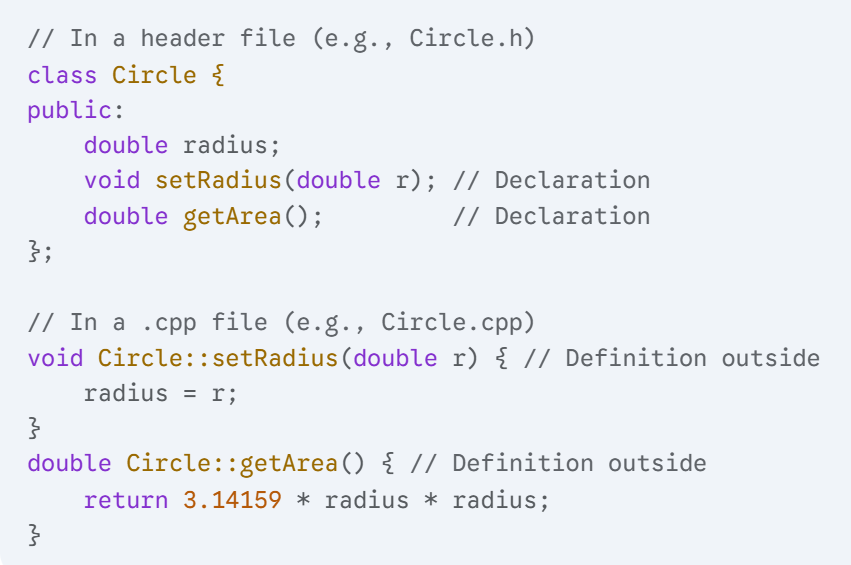
* Functions declared within a class are its member functions. They define the behaviors or operations that objects of the class can perform.

**Defining Member Functions:**

1. **Inside the Class Definition (Inline):**

****

**2.Outside the Class Definition (Non-inline):**

****

**Individual Copies:** "If Student s1; and Student s2; are two objects of the Student class, how many copies of the name data member (assuming name is not static) will exist?" (Options: One copy shared by both, **Two copies (one for s1, one for s2**), Zero copies, One copy in global memory)

### Constructors and Destructors

**Constructors:**

* A **constructor** is a special member function that is automatically called when an object of the class is created (instantiated).
* **Purpose:** To initialize the object's data members and/or allocate any resources (like dynamic memory) the object needs.
* **Copy Constructor:** A special constructor used to create a new object as a copy of an existing object. It takes a const reference to an object of the same class. (
* // Point p2 = p1; // Calls copy constructor
* // Point p3(p1); // Calls copy constructor
* C++ **does not have the concept of static constructors**.
* **Question 15: How will the objects be created if we don't have a constructor in a class of C++?** a) The compiler will generate error b) Error will occur at run-time c) Compiler provides its default constructor to build the object d) All variables of the class will automatically become garbage values.
* The correct option is **c) Compiler provides its default constructor to build the object**.

Question 16: Which of these is the most ideal way to initialize a member of a class in C++? a) Inside a constructor b) Outside the constructor by assigning values **c) Initialization List** d) All the above

**c) Initialization List (Most Ideal/Preferred):** This is the **most ideal and efficient** way to initialize member variables in C++. When you initialize members in the initializer list (e.g., MyClass(int val) : memberVar(val) {} or memberVar{val}), the member variable is **directly initialized** when the object is created. This is crucial for:

* const members: They *must* be initialized in the initializer list.
* Reference members: They *must* be initialized in the initializer list.
* Members of classes that don't have a default constructor.
* Efficiency: For complex objects, initializing in the list avoids a default construction followed by an assignment, which can be more efficient.

**Destructors:**

* A **destructor** is another special member function that is automatically called when an object is about to be destroyed (its lifetime ends).
*  Has the **same name as the class, prefixed with a tilde (~)**.
*  Has **no return type** (not even void).
*  Takes **no arguments** (cannot be overloaded).
*  Only one destructor per class.
*  Cannot be explicitly called by the programmer.

🡺What is the return type of a constructor?" (Options: void, int, The class type, No return type (implicitly void but not declared))

**Question 13: Which of the following statements are not true about destructor?**

1. It is invoked when object goes out of the scope
2. Like constructor, it can also have parameters
3. It can be virtual
4. It can be declared in private section
5. It bears same name as that of the class and precedes Lambda sign

a) Only 2, 3, 5 b) Only 2, 3, 4 c) Only 2, 1, 5 d) Only 3, 4, 5

Let's evaluate each statement:

1. **It is invoked when object goes out of the scope:** **TRUE**. This is one of the primary ways destructors are called (for stack-allocated objects). They are also called when a heap object is deleted.
2. **Like constructor, it can also have parameters:** **FALSE**. Destructors *cannot* have parameters. They are unique functions that take no arguments.
3. **It can be virtual:** **TRUE**. Destructors *can and often should* be virtual in polymorphic base classes to ensure that the correct derived class destructor is called when deleting an object through a base class pointer.
4. **It can be declared in private section:** **TRUE**. While less common for typical usage, a destructor can indeed be declared in the private section. This prevents external code from directly deleting objects (e.g., used in singleton patterns or for managing objects via factory methods).
5. **It bears same name as that of the class and precedes Lambda sign:** **FALSE**. It bears the same name as that of the class and precedes a **Tilde (~)** sign, not a Lambda sign ([] or ->).

### Array of Objects

 Just like you can create arrays of built-in types (like int or double), you can also create arrays of objects of a class. This allows you to manage a collection of objects conveniently.

 **Declaration:** ClassName arrayName[size];

**Important:** When dynamically allocating an array of objects with new ClassName[size], you *must* use delete[] arrayPointer; to deallocate it. Using delete arrayPointer; will only call the destructor for the first object and lead to undefined behavior/memory leaks for the rest.

**MCQ**

**Array of Objects Initialization:** "When an array of objects is created (e.g., MyClass objs[5];), which constructor is automatically called for each object?" (Options: Parameterized constructor, Copy constructor, **Default constructor**, Destructor)

**Quick Recap: Session 8**

* **Classes and Objects:** Classes are blueprints (class keyword), objects are instances of those blueprints. OOP principles like **encapsulation** (binding data and functions) are fundamental.
* **Access Specifiers (public, private, protected):** Control visibility of members. private is default for class. Data hiding is achieved by making data private and providing public member functions (getters/setters).
* **Defining Members:** Data members represent attributes. Member functions define behaviors and access class data. Functions can be defined inline (inside class) or outside using :: (scope resolution operator).
* **Constructors:** Special functions with class name, no return type, automatically called on object creation. Used for initialization. Types: Default, Parameterized, Copy.
* **Destructors:** Special functions ~ClassName(), no return type, no arguments, automatically called on object destruction. Used for cleanup (e.g., deallocating dynamic memory).
* **Array of Objects:** Collections of objects. Default constructor is called for each element on creation. Dynamic arrays of objects created with new ClassName[size] must be deallocated with delete[]. Access members using . operator for stack objects, -> for pointers to objects.

### 4. Namespaces

 **Namespaces** are a mechanism in C++ to group related entities (classes, functions, variables, enums, etc.) under a common name, thereby preventing **name collisions** in large projects or when using multiple libraries.

 **Purpose:** To organize code and avoid ambiguity when different libraries or different parts of a program use the same names for different entities.

**Example of Collision:** If LibraryA has a function print() and LibraryB also has a function print(), directly calling print() would be ambiguous. Namespaces resolve this.

**using namespace Directive:** Imports *all* names from a namespace into the current scope.

**std Namespace:** The entire C++ Standard Library (including std::cout, std::cin, std::string, etc.) resides within the std namespace. This is why you often see std:: prefix or using namespace std;.

MCQ

 **Accessing Namespace Members (Specific):** "To access myFunction() declared within namespace Utils, which of the following is the most specific and conflict-avoiding way?" (Options: myFunction();, using namespace Utils; myFunction();, Utils::myFunction();, Utils.myFunction();)

 **using namespace Directive:** "What is a potential drawback of using a using namespace directive, especially in header files?" (Options: It slows down compilation, It increases memory usage, It can reintroduce name collisions, It makes code less readable)

 **Standard Library Namespace:** "Most of the C++ Standard Library components (like cout, string) are located in which namespace?" (Options: main, global, standard, std)

🡺revison

**Overloading:**

* **Function Overloading:** Multiple functions with the same name but different parameter lists.
* **Operator Overloading:** Redefining C++ operators for user-defined types, using operator keyword, as member or friend functions.

 **Inheritance:**

* Allows a derived class to inherit from a base class ("is-a" relationship).
* Types (public, protected, private) affect accessibility of inherited members.
* Constructors call order: Base then Derived; Destructors: Derived then Base.

 **Polymorphism:** "Many forms."

* **Compile-time (Static):** Achieved by function overloading and operator overloading. Decisions at compile time.
* **Runtime (Dynamic):** Achieved by virtual functions and function overriding, accessed via base class pointers/references. Decisions at runtime.
* **virtual functions:** Allow specific derived class implementation to be called through a base pointer/reference.
* **Pure Virtual Functions (= 0;):** Make a class **abstract**, meaning it cannot be instantiated. Derived classes must implement them.

 **Namespaces:**

* Group related entities to prevent name collisions.
* Accessed using :: (scope resolution operator), using declaration (for specific names), or using namespace directive (for all names).
* std is the standard library namespace.

# Constructors and Destructors

### Parameterized Constructors

class Point {

public:

int x;

int y;

// Parameterized Constructor

Point(int initialX, int initialY) {

x = initialX;

y = initialY;

std::cout << "Parameterized constructor called for (" << x << ", " << y << ")" << std::endl;

}

};

**Ways to Call a Parameterized Constructor:**

**Direct Initialization (Explicit Call - preferred for clarity):**

Point p1(10, 20); // Calls Point(int, int)

1. **Copy-Initialization (Implicit Call - avoids if explicit conversion exists):**

// This syntax primarily uses copy constructors or move constructors

// but for single-argument constructors, it can also use parameterized constructor

// e.g., Point p = 5; // if Point has Point(int val)

// For multiple arguments, you generally need direct initialization.

1. **Uniform Initialization (Brace Initialization - C++11 onwards, highly recommended):**
   * Provides a consistent syntax for initialization and prevents narrowing conversions.

**Point p2{30, 40}; // Calls Point(int, int)**

* **Note** on Default Constructor Absence: If you define *any* parameterized constructor, the compiler will *not* automatically generate a default constructor. If you then try to create an object without arguments, it will be a **compilation error unless** you explicitly define a default constructor.

**// Point p3; // Error! No default constructor if only Point(int, int) exists.**

**MCQ**

Given class Item { public: Item(int id); };, which is a valid way to create an Item object with id 10 using its parameterized constructor?"

Options:

Item item;, Item item(10);,

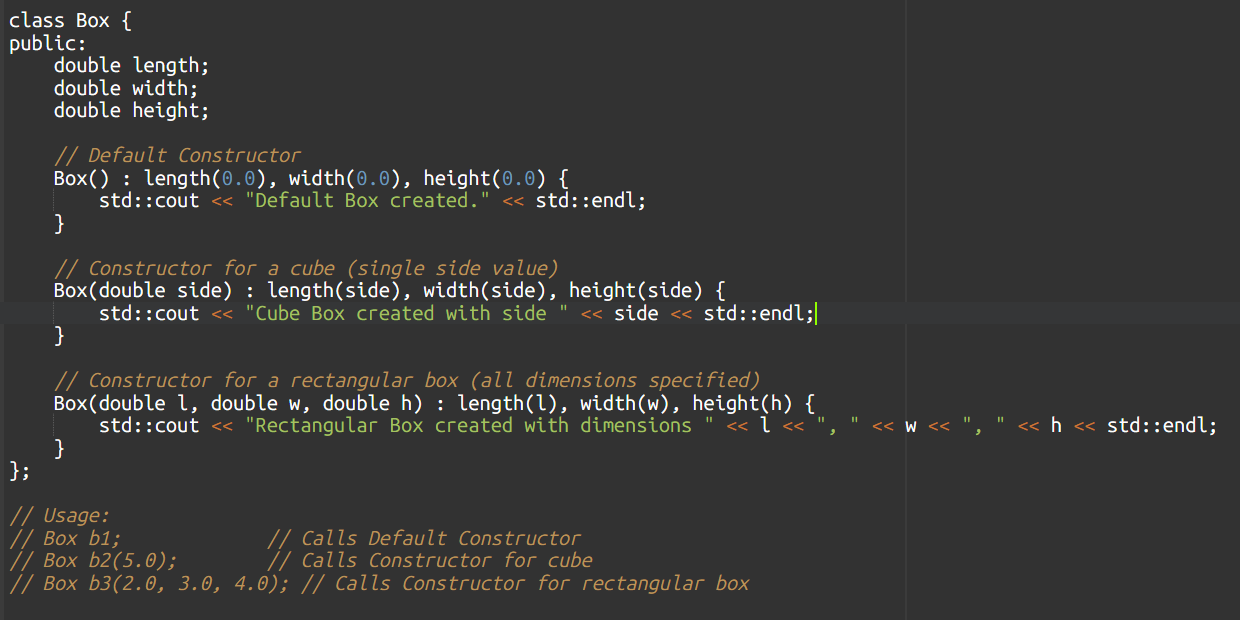
Item item = 10;, Item item{10};

Ans 🡺 (Note: both Item(10) and Item{10} are valid, so options might test subtle differences or focus on direct vs uniform init).

### Multiple Constructors in a Class (Constructor Overloading)

A class can have **multiple constructors**, provided each constructor has a unique **signature** (different number or types of parameters, or their order). This is a specific application of **function overloading**.

**Purpose:** To offer various ways to initialize objects, catering to different scenarios or levels of initial data availability.



**Delegating Constructors (C++11 onwards):** Allows one constructor to call another constructor of the same class to avoid code duplication.

Having multiple constructors in a class is an example of which C++ feature?" (Options: Inheritance, **Polymorphism (specifically function overloading)**, Encapsulation, Abstraction)

### Dynamic Initialization of Objects

**Dynamic initialization of objects** refers to creating objects during program execution (runtime) rather than at compile time (static initialization) or when a function scope is entered (automatic initialization). This involves using the new operator.

**Syntax:**

ClassName\* pointerName = new ClassName(arguments\_for\_constructor); // For single object

ClassName\* arrayPointer = new ClassName[size]; // For array of objects

**Interaction with Constructors:**

* When new ClassName(args) is used, the constructor matching args is called immediately after memory allocation on the heap.
* When new ClassName[size] is used, the **default constructor** is called for *each* object in the array

1. Copy Constructors

 A **copy constructor** is a special constructor that is used to create a new object as a **copy** of an already existing object of the same class.

 **Signature:** ClassName(const ClassName& other);

**Default Copy Constructor (Shallow Copy):**

* If you don't provide your own copy constructor, the compiler provides a **public, implicit copy constructor**.
* This default copy constructor performs a **member-wise (shallow) copy**. It copies the values of all data members from the source object to the destination object.

**Deep Copy (When to define your own):**

* When a class contains pointers to dynamically allocated memory, you need to implement a **deep copy**.
* A deep copy creates *new* memory for the copied object's pointer members and then copies the *contents* from the source's pointed-to memory.

MCQ

"What common problem does the default (shallow) copy constructor lead to in classes that manage dynamic memory?" (Options: Stack overflow, Compile-time errors, **Dangling pointers and double-free issues,** Infinite loops)

"In which of the following scenarios is a copy constructor typically called?" (Options: When creating an object with new, When returning an object by reference, **When initializing a new object from an existing one**, When calling a member function)

**Rule of Three/Five/Zero:** "If a class dynamically allocates memory, which set of special member functions should typically be explicitly defined to avoid issues like memory leaks and double-frees?" (Options: Constructors and Destructors only, Default and Parameterized Constructors, Copy Constructor**, Copy Assignment Operator, and Destructor**, All member functions)

### .6 Destructor

 A **destructor** is a special member function that is automatically invoked when an object goes out of scope or is explicitly destroyed (e.g., via delete for heap-allocated objects).

 **Purpose:** To perform necessary cleanup operations before the object's memory is reclaimed. This primarily involves **releasing resources** that the object might have acquired during its lifetime

**Virtual Destructor Importance:** "Why is it generally important to declare a base class destructor as virtual in a polymorphic hierarchy?" (Options: To allow overloading of the destructor, To prevent memory allocation**, To ensure the correct derived class destructor is called,** To make the class abstract)

When delete[] ptr; is called for a dynamically allocated array of objects, what happens to each object in the array?" (Options: Only the first object's destructor is called, Destructors are not called, **Each object's destructor is called**, A compile-time error occurs)

 "How many destructors can a class have in C++?" (Options: Zero, **One**, Multiple (based on overloading), As many as constructors)

**Quick Recap: Session 10 (Part 1)**

* **Constructors:** Special functions automatically called on object creation for initialization. No return type, same name as class.
* **Parameterized Constructors:** Take arguments to initialize objects with specific values.
* **Multiple Constructors (Overloading):** A class can have several constructors with different parameter signatures to offer varied initialization options.
* **Dynamic Initialization:** Creating objects on the heap at runtime using new. Requires delete to free memory. For arrays of objects (new ClassName[size]), the default constructor is called for each, and delete[] must be used.
* **Copy Constructors:** Special constructor ClassName(const ClassName& other) used to create a new object as a copy of an existing one. Called on initialization from another object, pass-by-value, and return-by-value.
* **Shallow vs. Deep Copy:** Default copy constructor performs shallow copy (member-wise copy). For classes with dynamic memory, this leads to issues (dangling pointers, double-free). A **deep copy** (programmer-defined copy constructor) is needed to allocate new memory and copy contents. This relates to the **Rule of Three/Five/Zero**.
* **Destructors:** Special functions ~ClassName() automatically called on object destruction for cleanup (releasing resources). No return type, no arguments, only one per class. Crucial for managing dynamically allocated memory.

# Inheritance – Extending Class

**Inheritance:** Allows code reuse, models "is-a" relationships (derived class extends base class).

**Types of Inheritance:**

* **Single:** One base class, one derived class.
* **Multiple:** One derived class, multiple base classes. Can lead to **Diamond Problem**.

**Key Challenge: The Diamond Problem (or Deadly Diamond of Death):**

* Occurs when a class inherits from two classes that, in turn, inherit from a common base class.
* **Problem:** If the common base class has data members or member functions, the derived class will have multiple copies or ambiguous access paths to them, leading to compilation errors or unexpected behavior.

**Solution (Virtual Base Class):** C++ provides **virtual base classes** to resolve the diamond problem (covered later in this session).

* **Multilevel:** Chain of inheritance (A -> B -> C).
* **Hierarchical:** One base class, multiple derived classes.
* **Hybrid:** Combination of any of the above.

### Virtual Base Class

1.  A **virtual base class** is a mechanism in C++ used to solve the **Diamond Problem** in multiple inheritance.
2.  When a base class is declared virtual during inheritance, it ensures that only **one copy** of the shared base class sub-object is inherited by the ultimate derived class, even if it is inherited through multiple paths.
3.  **Syntax:** The virtual keyword is placed *before* the access specifier in the derived class's inheritance list.

 **Chain of Calls:** In multilevel inheritance (e.g., Grandparent -> Parent -> Child), the constructors are called from the top of the hierarchy downwards: Grandparent -> Parent -> Child.

 **Order in Multiple Inheritance:** Constructors are called in the order of inheritance specified in the derived class's declaration.

class B1 { public: B1() { std::cout << "B1" << std::endl; } };

class B2 { public: B2() { std::cout << "B2" << std::endl; } };

class D : public B1, public B2 { public: D() { std::cout << "D" << std::endl; } };

// D obj; // Output: B1, B2, D

**Important:** You cannot call a base class constructor from within the *body* of the derived class constructor; it must be done in the **initializer list**.

"How does a derived class constructor pass arguments to its base class constructor?" (Options: In the function body, Using this pointer, **In the initializer list**, By defining a static method)

**Constructors in Derived Class:**

* Called in order: **Base then Derived**.
* Derived class constructor uses an **initializer list** to explicitly call a specific base class constructor and pass arguments.
* If no explicit call and base has a default constructor, the base's default constructor is called implicitly.

# Polymorphism

1. Types of Polymorphism

* 1. Compile-time Polymorphism (Static Polymorphism / Early Binding)

Achieved through:

* **Function Overloading:** Defining multiple functions with the same name but different parameter lists.
* **Operator Overloading:** Redefining the behavior of operators for user-defined data types.

It's "static" because the decision is made at a fixed point (compilation).

2. Runtime Polymorphism (Dynamic Polymorphism / Late Binding):

chieved primarily through:

* **virtual functions:** Functions declared in a base class with the virtual keyword and overridden in derived classes.
* **Base Class Pointers/References:** The call must be made through a pointer or reference to the base class type.

It's "dynamic" because the decision is made when the program is actually running.

**Member Function Overloading:** Functions within a class can also be overloaded.

class Calculator {

public:

int multiply(int x, int y) { return x \* y; }

double multiply(double x, double y) { return x \* y; }

};

1. Overloading Operators
   1. **Operator Overloading** allows you to redefine the behavior of C++ operators (e.g., +, -, \*, ==, [], <<, >>) when applied to user-defined data types (objects of your classes).

**Rules and Restrictions:**

1. You cannot create new operators.
2. You cannot change the precedence or associativity of operators.
3. You cannot change the arity (number of operands) of operators.
4. Cannot overload certain operators (e.g., . (dot), .\* (pointer to member), :: (scope resolution), ?: (ternary conditional), sizeof).
5. Friend Functions
   *  A **friend function** (or a **friend class**) is a non-member function (or an entire class) that is granted special permission to access the private and protected members of another class.
   *  **Declaration:** A friend function is declared inside the class whose members it needs to access, preceded by the friend keyword. It is *not* a member of that class.

**5. Constant Functions (Const Member Functions)**

* **Explanation (Clear & Deep):**
  + A **const member function** (often called a "constant function") is a member function that is declared with the const keyword appended to its signature.
  + **Purpose:** To indicate that the function will **not modify** the state (data members) of the object on which it is called.

**mutable Keyword:** The mutable keyword can be used with a non-static data member to allow it to be modified even within a const member function. This is rare and typically used for caching or logging purposes that don't change the logical state of the object.

Which keyword allows a data member to be modified even within a const member function?" (Options: static, volatile, friend, mutable)

**🔶 Quick Recap Table**

| **Concept** | **Type** | **Description** | **Binding Time** |
| --- | --- | --- | --- |
| Function Overloading | Compile-time | Same name, different params | Compile Time |
| Operator Overloading | Compile-time | Redefine operator behavior | Compile Time |
| Virtual Function | Runtime | Redefine function in derived class | Runtime |
| Friend Function | Not member | Can access private members | N/A |
| Const Function | Const qualifier | Cannot modify object state | N/A |

**Q5:** In v1 + v2, which object calls the operator function?  
A) v2  
B) Both  
C) v1  
D) None  
✅ **Answer:** C) v1

**Q6:** What’s the signature of a **postfix** operator++?  
A) operator++()  
B) operator++(int)  
C) operator++(float)  
D) ++operator()  
✅ **Answer:** B) operator++(int)

**Q8:** Why is operator<< usually a friend function?  
A) To overload unary operators  
B) Because cout is not a class  
C) Because the LHS is ostream, not class object  
D) To change operator precedence  
✅ **Answer:** C) Because the LHS is ostream, not class object

# 🎓 Session 12: Virtual Functions and Abstract Classes

### 1. Run Time Polymorphism

**Runtime Polymorphism** means deciding *which* function to run *during program execution* (not during compilation).

This is also called **Dynamic Polymorphism** or **Late Binding**

**🔶 2. Virtual Functions**

**🧠 Definition:**

A **virtual function** is a function in a **base class** that you expect to **override in derived classes**. When you declare a function as virtual, you allow **runtime binding** based on the object’s **actual type**, not the type of the pointer/reference.

**✅ Basic Rules:**

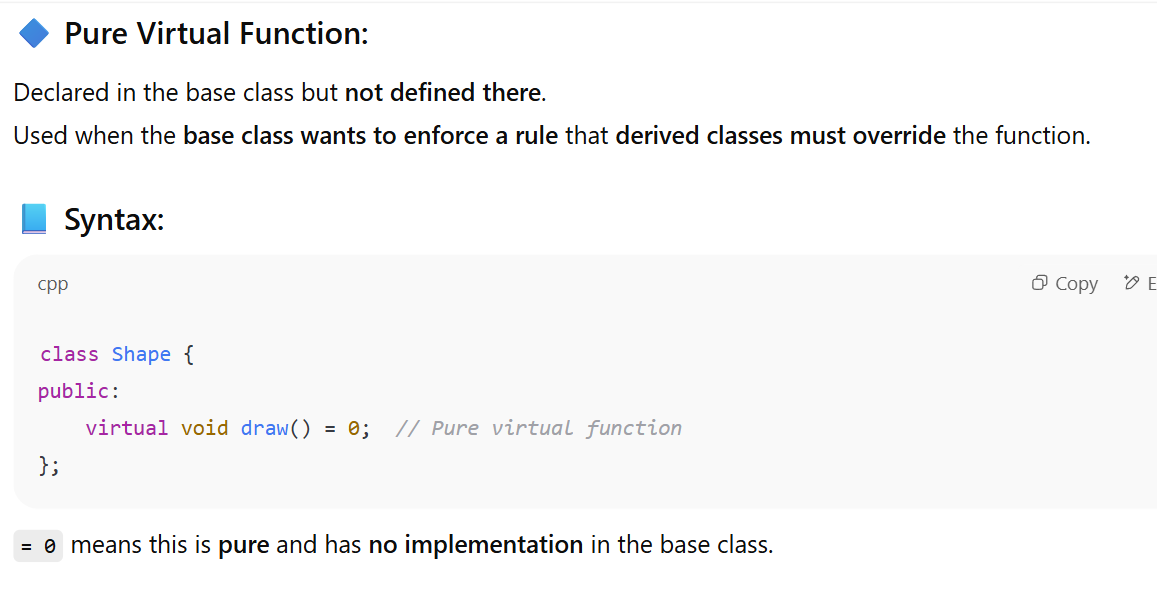
| **Rule** | **Explanation** |
| --- | --- |
| Use virtual keyword | In the base class only |
| Function must be overridden | In the derived class (optional) |
| Always use **base class pointer/reference** | For polymorphism to work |
| Must be a **non-static** member | Virtual functions cannot be static |



**❌ If sound() was not virtual:**

Output would be: **Animal makes a sound**

**Q1:** What is the purpose of the virtual keyword in C++?  
✅ To allow function overriding at runtime via base class pointers



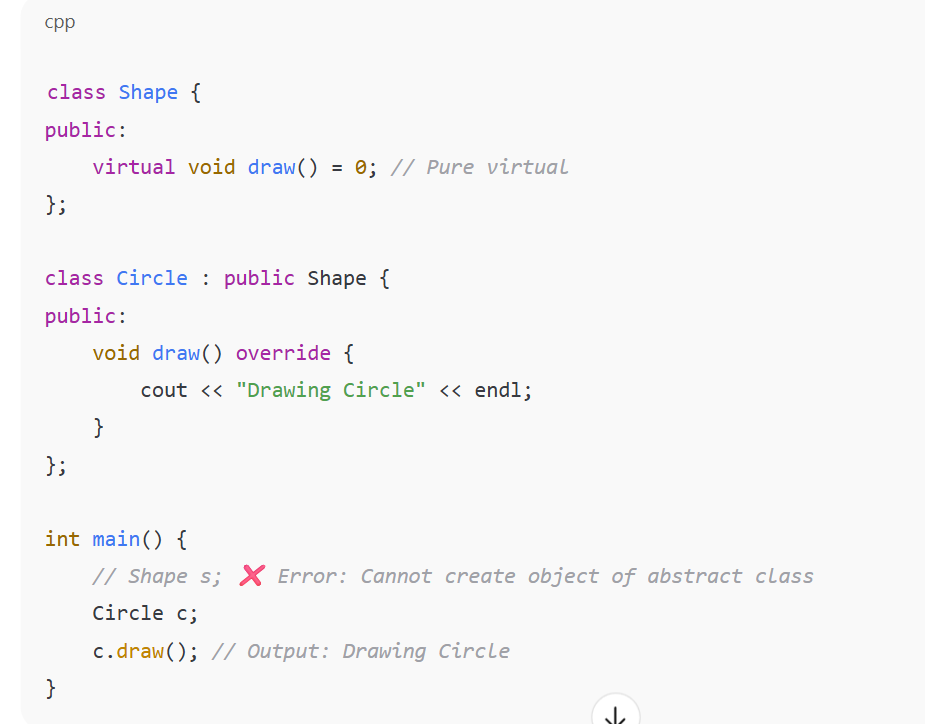
**🔷 Abstract Class:**

Any class with **at least one pure virtual function** is called an **abstract class**.

You **cannot create objects** of abstract classes.

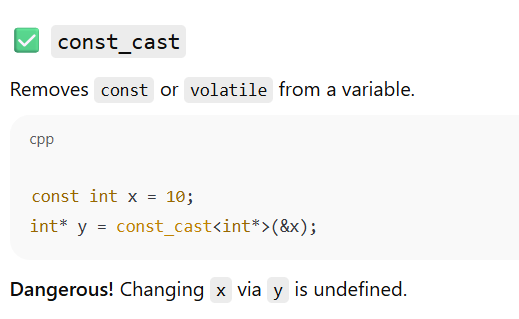
They are used to:

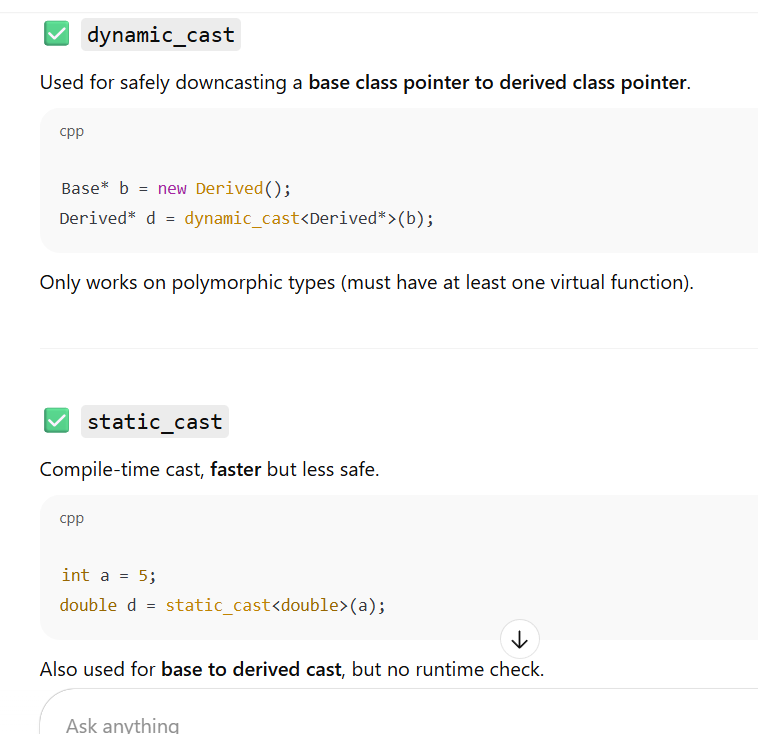
* Define **interfaces**
* Force **derived classes** to implement certain functions

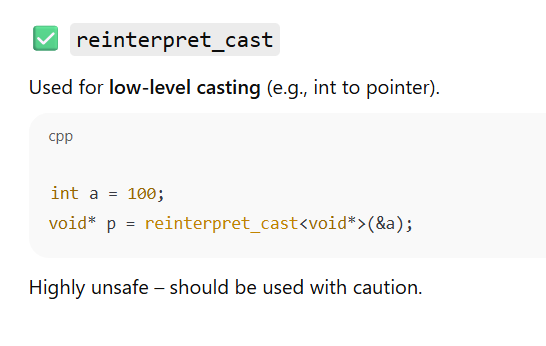


**🔶 4. dynamic\_cast, static\_cast, const\_cast, reinterpret\_cast**

These are **C++ casting operators** used to safely convert pointers or references.

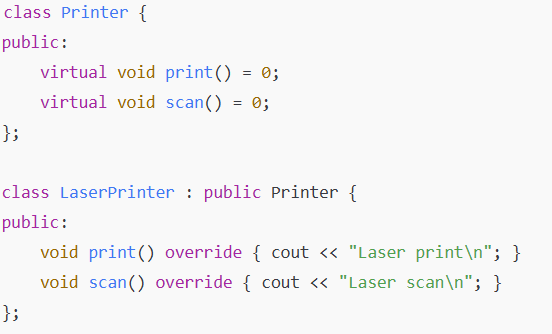


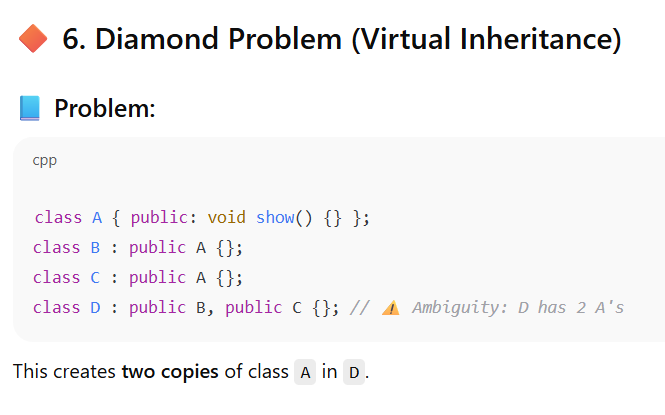


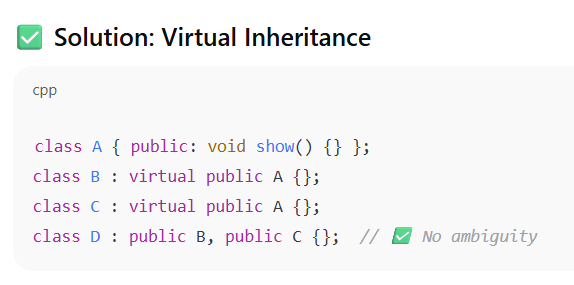


**🔶 5. Interface (via Abstract Class)**

C++ does not have a keyword like interface in Java.  
Instead, we use an **abstract class with all pure virtual functions**.







**✅ Summary – Key Points**

| **Concept** | **Use** |
| --- | --- |
| virtual function | Enables run-time polymorphism |
| Pure virtual function (= 0) | Forces derived class to override |
| Abstract class | Cannot be instantiated |
| dynamic\_cast | Safe downcasting |
| static\_cast | Fast but unsafe cast (int to double, void\* to MyClass\*) |
| const\_cast | Removes constness |
| reinterpret\_cast | Unsafe low-level cast |
| Virtual inheritance | Solves diamond problem |

🡺 **:** What does override keyword do? **A:** It confirms a derived class function intends to override a base class virtual function, helping catch errors if the signature doesn't match.

**MCQ-focused Learning:**

* **Q:** Which cast performs a runtime check for safe downcasting in polymorphic hierarchies? **A:** dynamic\_cast.
* **Q:** Which cast can remove the const qualifier from a pointer? **A:** const\_cast.
* **Q:** Which cast is used for general, sensible compile-time conversions? **A:** static\_cast.
* **Q:** Which cast is considered the most dangerous and is used for low-level, unrelated type conversions? **A:** reinterpret\_cast.

**MCQ-focused Learning:**

* **Q:** In C++, how is an interface typically implemented? **A:** Using a class with only pure virtual functions.
* **Q:** Can you create an object of an interface class? **A:** No.
* **Q:** What is the primary purpose of an interface? **A:** To define a contract or set of behaviors that derived classes must implement.

**Quick Recap: Session 12**

* **Runtime Polymorphism:** Decision made at runtime, enabled by virtual functions and base class pointers/references.
* **Virtual Functions:** Declared with virtual, allow derived class overrides to be called via base pointers.
* **Pure Virtual Functions:** virtual func() = 0;. No implementation in base; forces derived classes to implement.
* **Type Casting Operators:**
  + static\_cast: General, compile-time conversion (e.g., int to double, upcasting/downcasting without runtime check).
  + dynamic\_cast: Safe downcasting for polymorphic types (runtime check, returns nullptr on failure). Requires virtual functions.
  + const\_cast: Adds/removes const/volatile. Dangerous if original variable was truly const.
  + reinterpret\_cast: Low-level, unsafe conversion between unrelated types.
* **Interfaces:** Purely abstract classes (all functions are pure virtual) that define a contract for behavior without implementation. Cannot be instantiated.
* **Abstract Class:** A class with at least one pure virtual function. Cannot be instantiated. Serves as a blueprint, forcing derived classes to implement specific behaviors.

.

# 🎓 Session 13: Exception Handling

**🔷 1. What is Exception Handling?**

**✅ Definition:**

**Exception Handling** is a mechanism to detect, handle, and recover from **unexpected run-time errors** in a structured way, without crashing the program.

**📌 Why use it?**

| **Problem Type** | **Traditional Handling** | **With Exceptions** |
| --- | --- | --- |
| Division by zero | Manual if checks | throw an exception |
| File not found | Check return codes | try-catch mechanism |
| Invalid input | Nested if logic | Cleaner code |

**🔶 2. Try–Catch–Throw Mechanism**

**🔷 try block:**

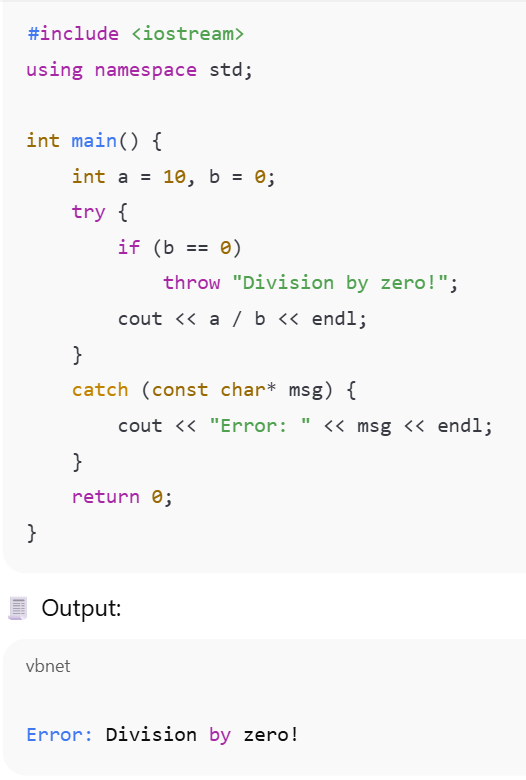
Contains code that **may cause an exception**

**🔷 throw statement:**

Used to **raise an exception**

**🔷 catch block:**

Catches the **exception object** and handles it

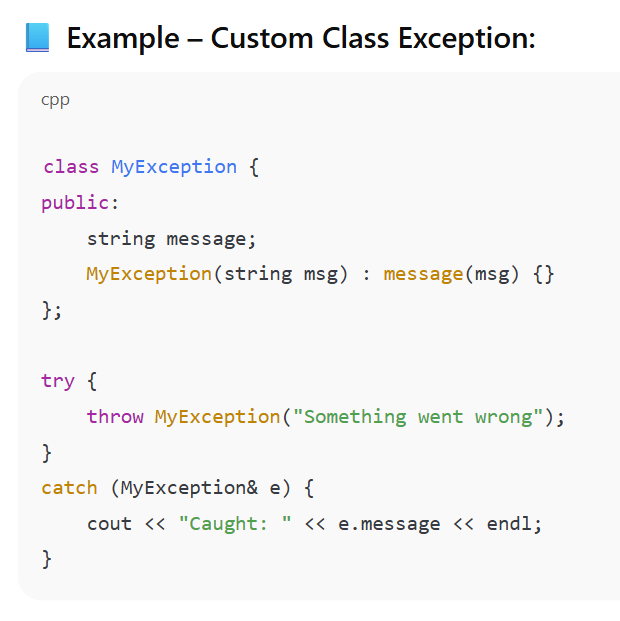


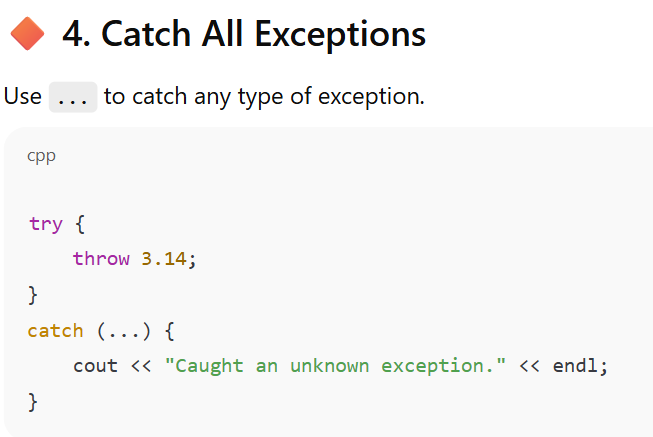
**🔁 Flow of Execution:**

1. Code inside try runs
2. If no exception → skips catch
3. If exception → jumps to catch

## 🔶 3. Types of Exceptions

| **Type** | **Example** |
| --- | --- |
| Built-in types | int, char, const char\* |
| Standard C++ types | std::exception, std::runtime\_error |
| User-defined classes | class FileError {} |





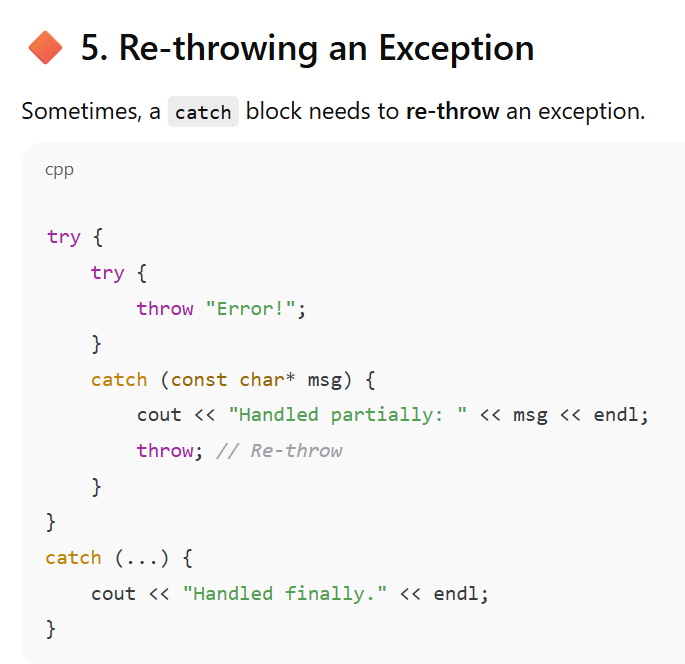
**Re-throwing an exception:** Sometimes, a catch block might handle part of an exception but needs to pass it on to a higher-level handler. You can **re-throw** the caught exception using throw; (without an argument) inside a catch block. This propagates the *original* exception object up the call stack.

🡺 What is the purpose of throw; inside a catch block? **A:** To re-throw the currently caught exception to a higher-level handler.

Mtlb hm catch block me throw ka use krte hai

**noexcept keyword:** Introduced in C++11 to indicate that a function will *not* throw any exceptions. If a noexcept function *does* throw an exception, std::terminate is called (usually crashing the program).

What happens if a function declared noexcept throws an exception? **A:** std::terminate is called (program usually crashes).



**Quick Recap: Session 13**

* **Exception Handling:** Mechanism to manage runtime errors using try, throw, and catch.
* **try block:** Code where exceptions might occur.
* **throw keyword:** Initiates an exception, passes an exception object.
* **catch block:** Handles exceptions by type. catch(...) catches all.
* **Re-throwing:** throw; inside catch sends the current exception up the call stack.
* **Exception Specifications (Old):** throw(type) to list possible exceptions (deprecated).
* **noexcept (Modern):** Guarantees a function won't throw; if it does, std::terminate is called. For functions that can throw, simply omit any specifier.

**🎓 Session 14: Managing Console I/O Operations**

🕒 **Duration**: 2 hours  
🎯 **Objective**: Understand how C++ handles input and output using stream-based I/O, including formatted and unformatted operations.

**🔷 1. Introduction to C++ I/O Streams**

C++ uses **streams** to perform input and output (I/O).

* A **stream** is a sequence of bytes.
* Input stream: Flow of data **into** the program (e.g., cin)
* Output stream: Flow of data **out of** the program (e.g., cout)

**🔧 I/O Objects:**

| **Object** | **Purpose** | **Header** |
| --- | --- | --- |
| cin | standard input (usually keyboard) | <iostream> |
| cout | standard output (usually console) | <iostream> |
| cerr | standard error (unbuffered) | <iostream> |
| clog | standard log/error (buffered) | <iostream> |

**🔶 2. Stream Classes**

I/O in C++ is based on a **class hierarchy** in the <iostream> library:

* **istream**: Input stream (used by cin)
* **ostream**: Output stream (used by cout)
* **iostream**: Combines both istream and ostream

**🔶 3. Unformatted I/O Operations**

Unformatted = Raw input/output without formatting

**🔹 Common Functions:**

| **Function** | **Description** |
| --- | --- |
| get() | Reads a single character |
| put() | Writes a single character |
| getline() | Reads a whole line into a string |
| read() | Reads binary data |
| write() | Writes binary data |

**📘 Example: get() and put()**

char ch;

cin.get(ch); // Read one character

cout.put(ch); // Print that character

**📘 Example: getline()**

char name[50];

cout << "Enter your full name: ";

cin.getline(name, 50); // Reads up to 49 characters or until newline

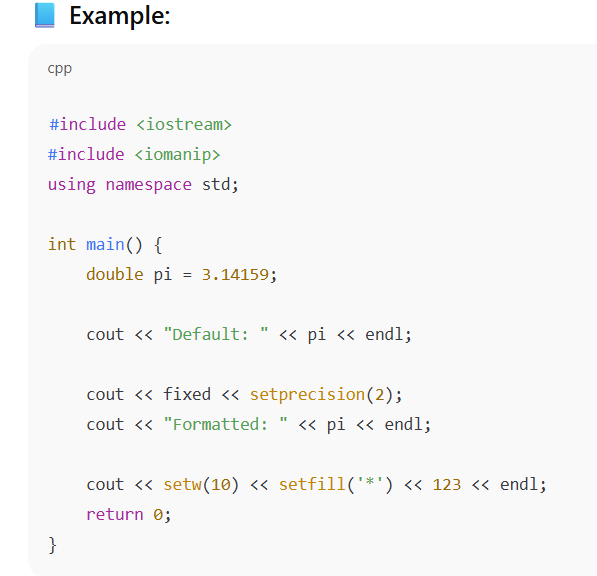
cout << "Welcome, " << name << "!";

**🔶 4. Formatted I/O Operations**

These operations allow **formatting** of the output using **manipulators**.

**🔹 Manipulators (from <iomanip>):**

| **Manipulator** | **Purpose** |
| --- | --- |
| setw(n) | Set field width |
| setfill(char) | Fill empty spaces |
| setprecision(n) | Set decimal precision |
| fixed | Fixed-point notation |
| scientific | Scientific notation |
| left, right | Alignment |

****

🧾 Output:

Default: 3.14159

Formatted: 3.14

\*\*\*\*\*\*\*123

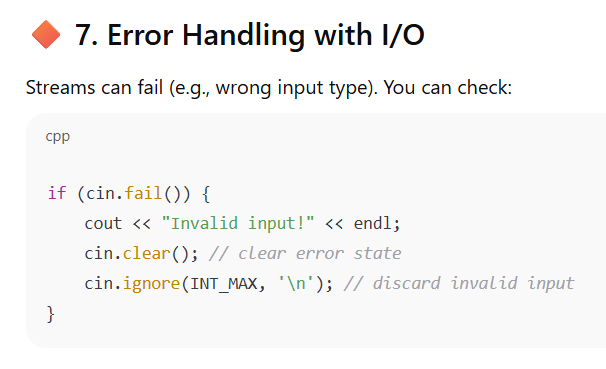
**🔶 5. Chaining I/O Operations**

You can chain output or input:

cout << "Name: " << name << ", Age: " << age << endl;

**🔶 6. cin vs. getline**

| **Feature** | **cin** | **getline()** |
| --- | --- | --- |
| Stops at | space/tab/newline | only newline |
| Can read spaces? | ❌ No | ✅ Yes |
| Syntax | cin >> str; | cin.getline(str, size); |

****

 **Q:** From which base class do istream and ostream inherit I/O state flags? **A:** ios.

 **Q:** Which stream class provides output operations like <<? **A:** ostream.

 **Q:** Which header file is typically included for console I/O operations? **A:** <iostream>.

 **Q:** Which function is typically used to read a single character from std::cin? **A:** get().

 **Q:** Which function is best suited for reading an entire line of text into a std::string? **A:** std::getline().

 **Q:** What does "unformatted I/O" imply? **A:** Reading/writing raw data without special formatting.

 **Q:** Which operator is used for formatted output to std::cout? **A:** << (stream insertion operator).

 **Q:** Which operator is used for formatted input from std::cin? **A:** >> (stream extraction operator).

 **Q:** What does the >> operator typically do with leading whitespace? **A:** It skips it.

 **Q:** What is the purpose of std::setw(int w)? **A:** To set the field width for the next output item.

 **Q:** Which header file contains most manipulators like setw and setprecision? **A:** <iomanip>.

 **Q:** What does std::endl do? **A:** Inserts a newline and flushes the output buffer.

 **Q:** How do you display a bool value as "true" or "false" instead of "1" or "0"? **A:** Use std::boolalpha.

**Quick Recap: Session 14**

* **Console I/O:** Input from keyboard (std::cin), output to screen (std::cout, std::cerr, std::clog).
* **Streams:** Abstract flow of bytes. istream for input, ostream for output.
* **Unformatted I/O:** Raw byte/character operations (get(), put(), read(), write(), getline()).
* **Formatted I/O:** Uses >> (extraction) for input and << (insertion) for output, converting types automatically.
* **Manipulators:** Special functions/objects (from <iomanip>) used with << to control output formatting (e.g., setw, setprecision, fixed, hex, boolalpha, endl).

# 🎓 Session 15: File Handling in C++

### 🔷 1. Introduction to Files in C++

Files are used for **persistent storage** — data that lives even after the program ends.

C++ provides file handling through the **fstream library**.

**Q:** What is a file primarily used for in computing? **A:** To store persistent data.

**Q:** What's the difference between a text file and a binary file? **A:** Text files store human-readable characters; binary files store raw bytes.

**📚 2. Required Header: <fstream>**

To work with files, include:

#include <fstream>

C++ provides 3 file stream classes:

| **Class** | **Purpose** |
| --- | --- |
| ifstream | Input file stream (read from file) |
| ofstream | Output file stream (write to file) |
| fstream | Input + Output file stream |

What is the first step when performing file I/O in C++? **A:** Include <fstream> and declare a stream object.

**🔹 3. Opening and Closing a File**

**✅ Using Constructors:**

ifstream fin("input.txt"); // Open file for reading

ofstream fout("output.txt"); // Open file for writing

**✅ Using open() method:**

fstream file;

file.open("data.txt", ios::in | ios::out); // Open for read + write

**✅ Closing:**

file.close(); // Always close files after use

Which of the following methods is used to close a file stream in C++? a) flush() b) end() c) **close**() d) terminate()

**🔹 4. File Opening Modes (Flags from ios)**

| **Mode** | **Meaning** |
| --- | --- |
| ios::in | Open for reading |
| ios::out | Open for writing |
| ios::app | Append to end of file |
| ios::ate | Move to end of file after opening |
| ios::trunc | Truncate file if it exists |
| ios::binary | Open file in binary mode |

🡺You can combine them using bitwise OR (|):

file.open("data.txt", ios::in | ios::out | ios::binary);

**ios::in :** If the file does not exist, opening it with ios::in will typically **fail (>>)**. (In contrast, ios::out would create it).(<<)

**Q:** Which header file is necessary for C++ file handling? **A:** <fstream>.

**Q:** Which class is used to write data to a file? **A:** ofstream.

**Q:** What happens if you open an ofstream with std::ios::out mode and the file already exists? **A:** The file's existing content is truncated (deleted).

**Q:** Which file opening mode ensures that new data is added at the end of the file? **A:** std::ios::app.

* + **Q:** Which function is commonly used to read an entire line from an ifstream into a std::string? **A:** std::getline().
  + **Q:** For reading/writing raw data (like an int directly as bytes) to a file, what mode should the file be opened in? **A:** std::ios::binary.

**🔹 5. Writing to a File**

#include <fstream>

using namespace std;

int main() {

ofstream fout("example.txt");

fout << "Hello, File!" << endl;

fout.close();

return 0;

}

Which operator is used to write formatted data to an ofstream object? **A:** <<

**🔹 6. Reading from a File**

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

int main() {

ifstream fin("example.txt");

string line;

while (getline(fin, line)) {

cout << line << endl;

}

fin.close();

return 0;

}

**🔹 7. Checking File Open/Fail**

ifstream fin("myfile.txt");

if (!fin) {

cout << "File couldn't be opened!" << endl;

return 1;

}

How do you check if a file stream object (e.g., outFile) has successfully opened a file? **A:** if (outFile.is\_open()) or if (!outFile.fail()).

**🔹 8. Binary File I/O**

C++ allows you to read/write **objects** in binary form:

cpp

CopyEdit

#include <fstream>

using namespace std;

class Student {

public:

int id;

char name[20];

};

int main() {

Student s1 = {1, "Shivam"};

ofstream fout("data.bin", ios::binary);

fout.write((char\*)&s1, sizeof(s1));

fout.close();

return 0;

}

To read:

cpp

CopyEdit

Student s2;

ifstream fin("data.bin", ios::binary);

fin.read((char\*)&s2, sizeof(s2));

**🔹 9. File Pointers**

Used to move within a file (important for random access):

| **Function** | **Purpose** |
| --- | --- |
| seekg() | Set input (get) position |
| seekp() | Set output (put) position |
| tellg() | Get input (get) position |
| tellp() | Get output (put) position |

cpp

CopyEdit

fin.seekg(0); // Move to start

fin.seekg(0, ios::end); // Move to end

int pos = fin.tellg(); // Current position

To read a number (int x;) from a text file input.txt using an ifstream object named inFile, which is the correct syntax?

a) inFile.read(&x, sizeof(x));

**b) inFile >> x;**

c) x = inFile.get();

d) inFile.getline(x);

exp🡺🡺 **inFile >> x;**: This uses the **extraction operator (>>)**, which is overloaded for various data types (like int, double, float, std::string, etc.).

**a) inFile.read(&x, sizeof(x));**: The read() member function is primarily used for **unformatted (binary) input**. It reads a specified number of bytes directly into a memory location. While it can technically read bytes that happen to represent an int in binary, it will not correctly parse a human-readable number like "123" from a text file.

**c) x = inFile.get();**: The get() member function reads a **single character** from the stream. If you read the character '1', its ASCII value (e.g., 49) would be stored in x, not the integer 1. It wouldn't handle multi-digit numbers like "123".

**d) inFile.getline(x);**: The getline() function is used to read an entire line of text (until a newline character or a delimiter is encountered) into a character array (char\*) or a std::string. It is not designed to directly parse an integer.



**Answer: b) Unformatted binary writing.**

* **Explanation:** write() is an unformatted function, and the file is opened in std::ios::binary mode, indicating a binary operation.

**When reading from a file, how can you determine if you have reached the end of the file?**

**Answer: b) By checking stream.eof().**

* **Explanation:** eof() (end-of-file) returns true when the end of the file has been reached during an input operation. It's often used in conjunction with other state checks (like good() or fail()) for robust error handling.

**5.** Which function moves the file pointer for input?  
a) seekp()  
b) move()  
c) seekg()  
d) moveg()  
✅ **Answer**: c) seekg()

# 🎓 Session 16: Templates in C++

**🔷 1. Why Templates?**

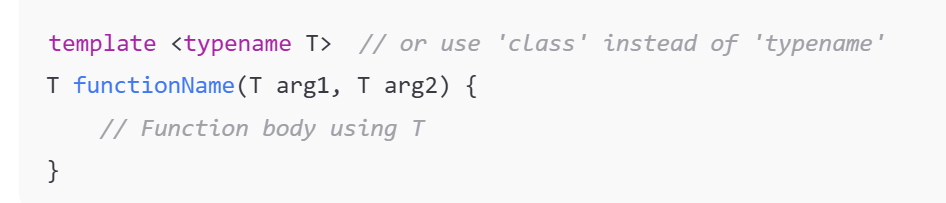
Templates allow you to write a **single function or class** that works with **any data type**, instead of duplicating code for each type.

Imagine writing separate addInt(int, int) and addFloat(float, float) functions. Templates let you write just **one** generic version!

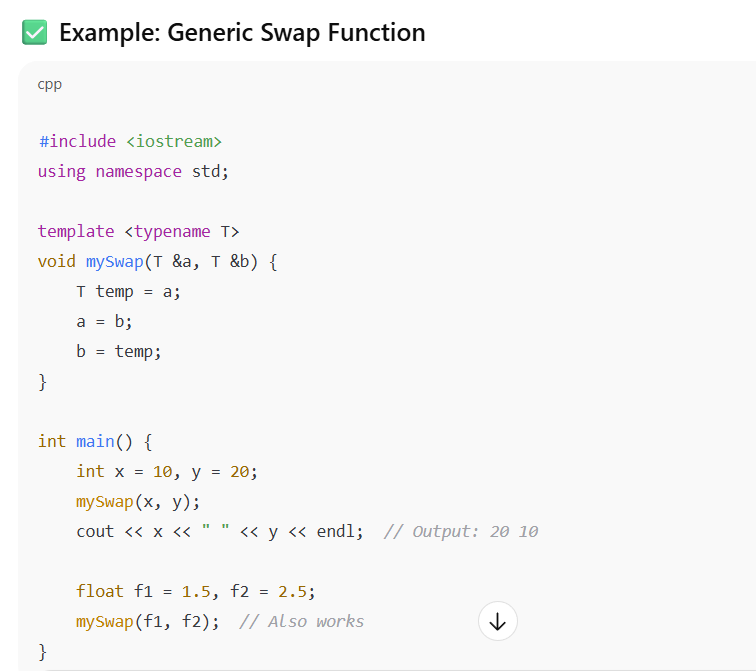
**🔹 2. Function Templates**

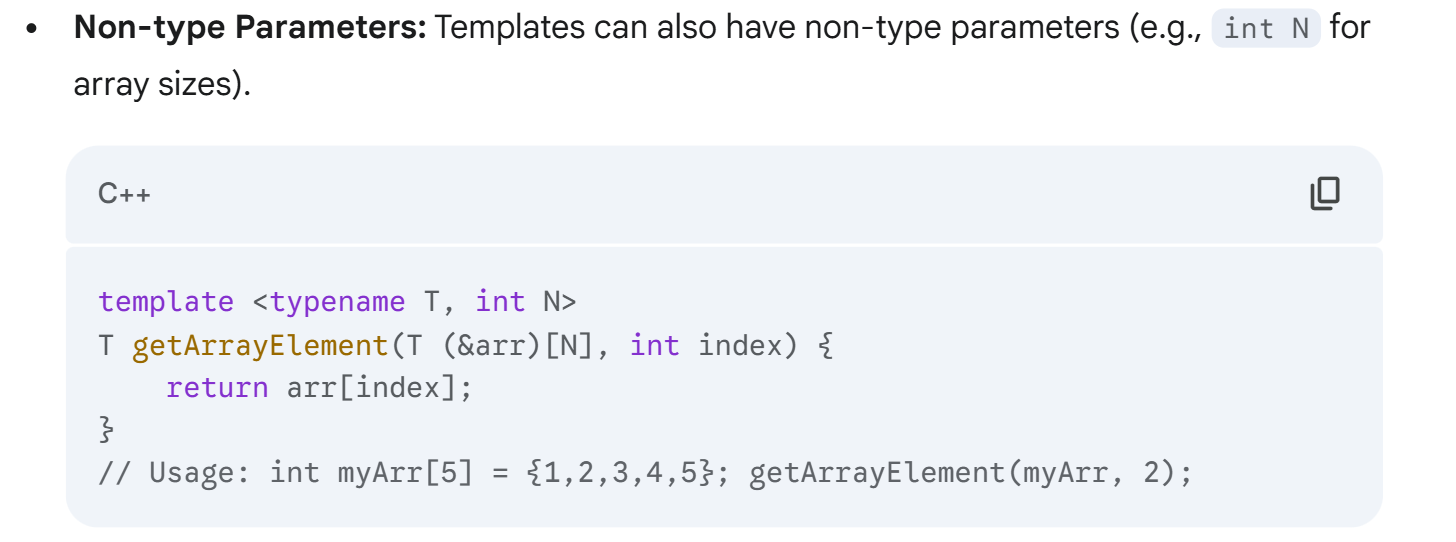
A **function template** is a blueprint for creating functions that can work with **any type**.

**✅ Syntax:**



**✅ Example: Generic Swap Function**





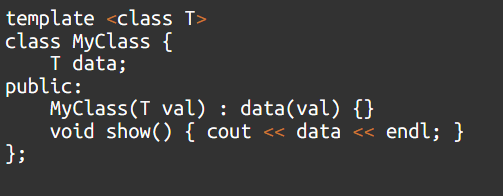
**Q:** When is a specific version of a function template generated? **A:** At compile time, when the function is called with a specific data type.

Can a function template have multiple type parameters? **A:** Yes (e.g., template <typename T1, typename T2>).

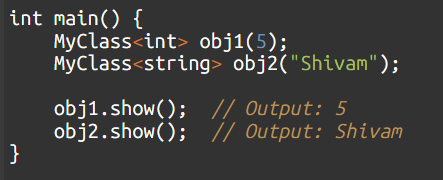
**🔹 3. Class Templates**

A **class template** defines a class where **data type is a parameter**.

Synatx

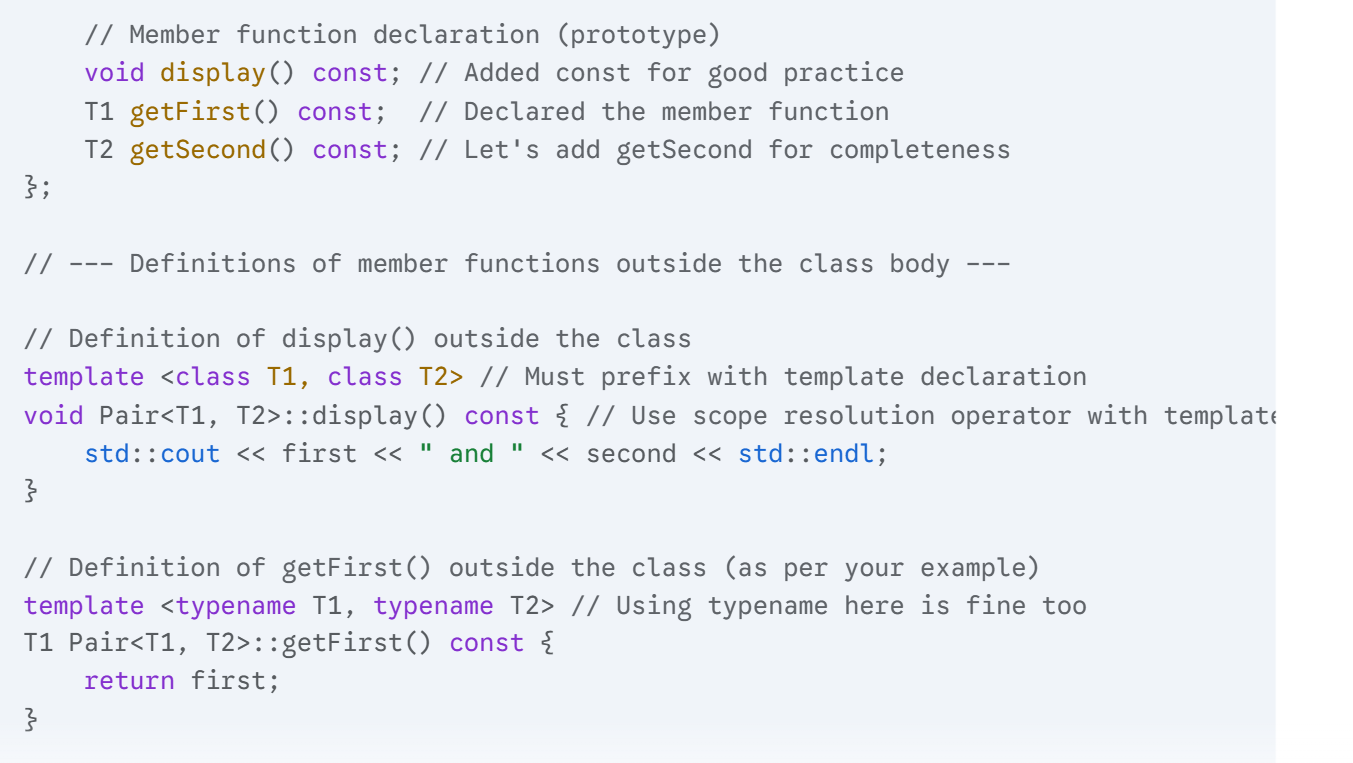


Example



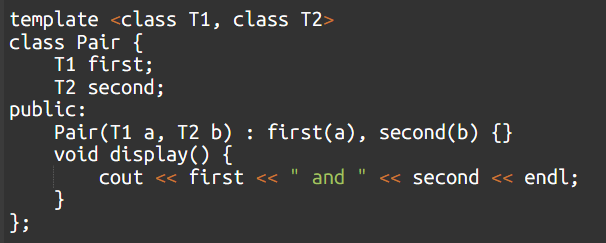
**Defining Member Functions Outside the Class:**

* When defining member functions of a class template outside the class body, you *must* prefix them with the template declaration.
*  **Q:** How do you declare an object of a class template (e.g., a Stack that holds ints)? **A:** Stack<int> myStack;.
*  **Q:** Can a class template have non-type parameters (e.g., for fixed-size arrays)? **A:** Yes (e.g., template <typename T, int N>



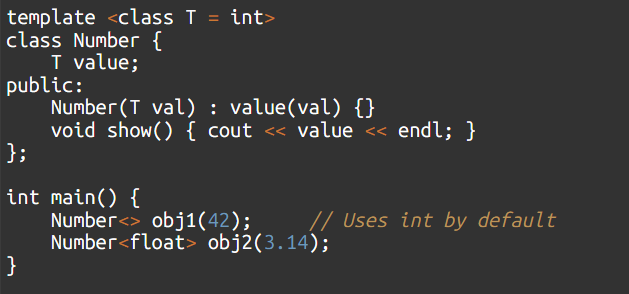
**🔹 4. Templates with Multiple Types**

You can define more than one type parameter:



**🔹 5. Templates and Default Arguments**

Templates can use **default type parameters**:



}

**🔹 6. Advantages of Templates**

| **Benefit** | **Description** |
| --- | --- |
| Generic Code | Write once, use with any type |
| Type Safety | Errors caught at compile-time |
| Reusability | No need to duplicate functions for every type |
| Performance | No overhead — compiler generates actual code for used types |

**⚠️ 7. Important Limitations**

* Templates **must be used in the same file** (or header files), because compiler needs to see the entire definition.
* Not all types may support all operations (e.g., + for objects not overloaded).
* **Template instantiation** only happens when the template is used.

**🧪 MCQ-Focused Learning**

Consider the following function template:

template <typename U>

U getMax(U a, U b) {

return (a > b) ? a : b;

}

If you call getMax(5, 10);, what type will U be instantiated as by the compiler?

a) double

b) float

c) **int**

d) char

🡺When defining a member function of a class template *outside* the class definition, what must you prefix the definition with? a) void b) public: **c) template <typename** T**> (assuming** T is the class's type parameter) d) inline

What happens if you try to use a function template with a data type for which the operations inside the template are not valid (e.g., comparing two objects of a custom class that doesn't have operator>)?

a) The compiler will automatically provide a default comparison.

b) A runtime error will occur.

c) A compilation error will occur.

d) The template will silently ignore the operation.

**Answer: c) A compilation error will occur.**

* **Explanation:** Templates are resolved at compile time. If the operations within the template are not valid for the substituted type, the compilation will fail.

🡺imp When you declare an object of a class template, e.g., MyList<int> list1;, what is MyList<int> called? a) A template definition. b) A template declaration. c) A template instantiation. d) A template parameter.

**Answer: c) A template instantiation.**

* **Explanation:** When you provide concrete types for a template, the compiler generates a specific class (or function) from the template, and this process is called instantiation.

**Can templates have non-type parameters** (e.g., integer values)?

a) No, templates can only have type parameters.

b) Yes, but only for class templates.

c) **Yes**, for both function and class templates (e.g., template <typename T, int N>).

d) Yes, but they must be const variables.

# 🎓 Session 17 & 18: STL (Standard Template Library) and RTTI (Run-Time Type Information)

**🔷 PART 1: Standard Template Library (STL)**

The **STL** is a powerful set of **template classes and functions** for managing data structures and performing common operations efficiently.

**✳️ STL Core Components:**

1. **Containers** – Store collections of data.
2. **Algorithms** – Perform operations like sort, search, etc.
3. **Iterators** – Work like pointers to access container elements.

**🔹 1. Containers**

A container is a class template that stores **objects** in memory.

**🧠 STL Containers Comparison Table (For MCQs & Interviews)**

| **Type** | **Container** | **Description** | **Performance Notes (Insertion/Deletion/Search Time Complexities)** |
| --- | --- | --- | --- |
| **Sequence** | vector | Dynamic array (most used) | ✅ Fast random access: O(1) 🔁  ❌ Insertion/deletion at **end**: O(1) (amortized)  ❌ At **middle/start**: O(n) |
| **Sequence** | list | Doubly-linked list | ✅ Insertion/deletion at **start/middle/end**: O(1) (if iterator is given)  ❌ No random access  🔍 Search: O(n) |
| **Sequence** | deque | Double-ended queue | ✅ Fast insertion/deletion at **both ends**: O(1)  🔍 Random access: O(1)  ❌ Insertion in middle: O(n) |
| **Associative** | set | Unique elements in sorted order | 🔍 Search, insert, delete: O(log n) (uses balanced BST - Red-Black Tree)  ❌ No duplicates |
| **Associative** | map | Key-value pairs | 🔍 Search, insert, delete by key: O(log n)  ❌ No duplicate keys |
| **Associative** | multiset / multimap | Allow duplicate keys | 🔍 All operations: O(log n)  ✅ Allows duplicates (sorted) |
| **Container Adaptor** | stack (uses deque or vector) | LIFO structure | ✅ Push/pop from top: O(1)  ❌ No traversal/search |
| **Container Adaptor** | queue (uses deque) | FIFO structure | ✅ Push at back, pop from front: O(1)  ❌ No traversal/search |
| **Container Adaptor** | priority\_queue (uses heap) | Max-heap (default) | ✅ Insert: O(log n)  ✅ Top (max): O(1)  ❌ Search not allowed |

**🧪 MCQ-STYLE NOTES & TIPS:**

1. **Which STL container gives fastest random access?** → vector, deque (O(1))
2. **Which container allows efficient insertion at both ends?** → deque
3. **Which container is best for frequent insertions/deletions at any position?** → list
4. **Which container stores sorted unique elements?** → set
5. **Which associative container allows duplicate keys?** → multimap, multiset
6. **Underlying data structure for map, set, etc.?** → Red-Black Tree (Self-balancing BST)
7. **Underlying data structure for priority\_queue?** → Max Heap
8. **Which STL container is best for key-value pair storage and fast lookup?** → map

**✅ Example: vector Usage**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> v = {1, 2, 3};

v.push\_back(4); // Add at end

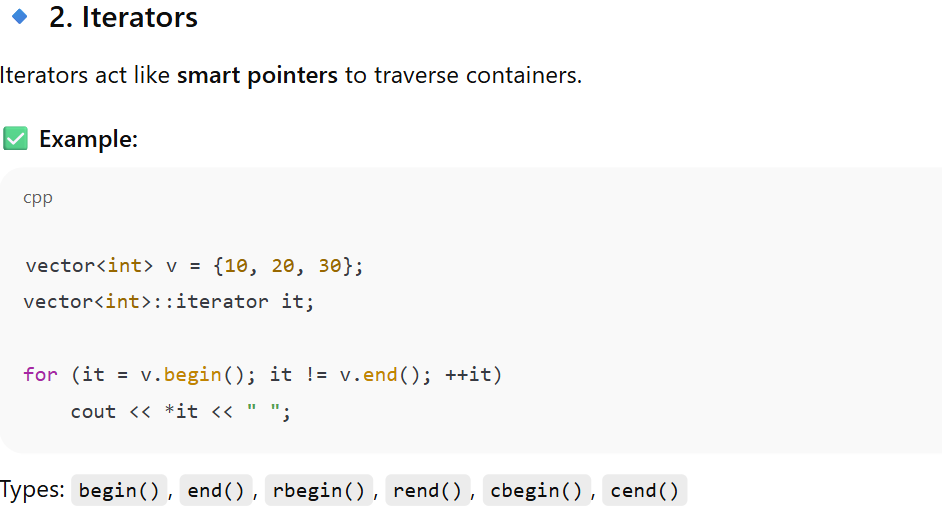
v.pop\_back(); // Remove last element

for (int i : v)

cout << i << " "; // Output: 1 2 3

return 0;

}



**📘 Iterator Functions**

| **Function** | **Direction** | **Returns** | **Use Case** | **Const?** | **Typical MCQ Tip** |
| --- | --- | --- | --- | --- | --- |
| begin() | Forward | Iterator to **first** element | Start of container traversal | ❌ | \*container.begin() is the 1st element |
| end() | Forward (past end) | Iterator **after last** element | Used in loops; not dereferenceable | ❌ | end() ≠ last element! Be careful. |
| rbegin() | Reverse | Iterator to **last** element | Reverse traversal (last to first) | ❌ | rbegin() starts from back |
| rend() | Reverse (past front) | Iterator **before first** element | End of reverse traversal | ❌ | Used with rbegin() in reverse loops |
| cbegin() | Forward | **Const iterator** to first | When you don't want to modify elements | ✅ | For const correctness |
| cend() | Forward | **Const iterator** after last | Like end(), but for read-only loops | ✅ | Only read, no write allowed |

**Q1. What does \*container.end() return?**  
a) Last element  
b) Garbage value  
c) First element  
d) Compilation error  
**✅ Answer:** ❌ *Invalid!* end() **cannot be dereferenced**

**🔹 3. Algorithms**

Header: <algorithm>

Some common STL algorithms:

* sort()
* find()
* count()
* reverse()
* accumulate() (from <numeric>)

**✅ Example:**

cpp

CopyEdit

#include <algorithm>

vector<int> v = {5, 3, 9, 1};

sort(v.begin(), v.end()); // Sorts in ascending order

**🔹 4. Associative Containers**

**✅ Example: map**

cpp

CopyEdit

#include <map>

map<string, int> age;

age["Shivam"] = 22;

age["Raj"] = 25;

cout << age["Shivam"]; // Output: 22

**✅ Example: set**

cpp

CopyEdit

#include <set>

set<int> s = {1, 2, 3, 2};

for (int i : s)

cout << i << " "; // Output: 1 2 3 (no duplicates)

**⚠️ Common STL Interview/MCQ Tips:**

* **STL containers are implemented using templates**.
* vector offers random access (like arrays), list does not.
* set elements are sorted by default and unique.
* map stores key-value pairs, keys must be unique.
* Iterators must be **dereferenced** (\*it) to access value.

**🧪 MCQ-Focused Practice (STL)**

**Q1.** Which STL container maintains elements in sorted and unique order?  
a) vector  
b) list  
c) set  
d) map  
✅ **Answer**: c) set

**Q2.** What is the purpose of an iterator in STL?  
a) Modify class structure  
b) Traverse through a container  
c) Generate random values  
d) Optimize I/O  
✅ **Answer**: b) Traverse through a container

**Q3.** Which STL container allows duplicate keys?  
a) map  
b) set  
c) multimap  
d) vector  
✅ **Answer**: c) multimap

**Q4.** What does sort(v.begin(), v.end()) do?  
a) Reverses the vector  
b) Prints elements  
c) Sorts the vector in ascending order  
d) Sorts the vector in descending order  
✅ **Answer**: c) Sorts the vector in ascending order

**Q5.** Which header file is required for STL containers?  
a) <iostream>  
b) <vector>  
c) <stl>  
d) <stdlib.h>  
✅ **Answer**: b) <vector> (or respective container’s header)

**🔷 PART 2: RTTI (Run-Time Type Information)**

**🔹 What is RTTI?**

**RTTI** (Run-Time Type Information) is a C++ feature that allows a program to determine the type of an object during **runtime** (while the program is executing).

It's primarily used in **polymorphic hierarchies** (classes with at least one virtual function). **base class pointers to derived objects).**

**✅ Useful RTTI Operators:**

| **Operator** | **Description** |
| --- | --- |
| typeid | Returns type of an object |
| dynamic\_cast | Safely downcasts a pointer/reference in class hierarchy |

**🔹**

**1. typeid Operator**

**typeid operator:** Returns a std::type\_info object, which contains information about a type (e.g., its name).

* **Header:** <typeinfo>
* **Usage:** typeid(object) or typeid(\*pointer).
* typeid on a polymorphic object (via a pointer/reference to a base class) will return the *actual derived type*.
* typeid on a non-polymorphic object or a null pointer will return the *static (declared) type*.
* name() member function of std::type\_info gives a string representation of the type name (may be mangled).
* Which operator is used to retrieve type information at **runtime**? **A:** typeid.

#include <iostream>

#include <typeinfo>

using namespace std;

int main() {

int x = 5;

cout << typeid(x).name(); // Output: int (may vary)

}

**With polymorphism:**

class Base { virtual void foo() {} };

class Derived : public Base {};

Base\* b = new Derived;

cout << typeid(\*b).name(); // Output: Derived (not Base)

**🔹 2. dynamic\_cast**

Used for safe **downcasting** in class hierarchies.

**dynamic\_cast operator:** (Revisit) Used for safe downcasting in polymorphic class hierarchies. If the cast is invalid, it returns nullptr (for pointers) or throws std::bad\_cast (for references). This is the most common and safer way to use RTTI.

**✅ Example:**

class Base {

public:

virtual void show() {}

};

class Derived : public Base {

public:

void derivedFunc() {}

};

Base\* basePtr = new Derived;

Derived\* dPtr = dynamic\_cast<Derived\*>(basePtr);

if (dPtr != nullptr) {

dPtr->derivedFunc(); // Safe to call

} else {

cout << "Cast failed";

}

**Q6.** Which operator is used to safely downcast a pointer in polymorphism?  
a) static\_cast  
b) reinterpret\_cast  
c) const\_cast  
d) dynamic\_cast  
✅ **Answer**: d) dynamic\_cast

**Q7.** Which header file is required for typeid?  
a) <iostream>  
b) <typeinfo>  
c) <rtti>  
d) <string>  
✅ **Answer**: b) <typeinfo>

**Q8.** RTTI is mainly used in:  
a) Compile time function resolution  
b) File I/O  
c) Dynamic casting and type checking  
d) Pointer arithmetic  
✅ **Answer**: c) Dynamic casting and type checking

**Quick Recap: Sessions 17 & 18**

* **C++ Standard Library:** Pre-built collection of components, promoting code reusability.
* **STL (Standard Template Library):** Part of the Standard Library, providing generic **containers** (data structures), **algorithms**, and **iterators**.
* **Common STL Containers:**
  + std::vector: Dynamic array, efficient random access.
  + std::stack: LIFO adaptor (push, pop, top).
  + std::queue: FIFO adaptor (push, pop, front, back).
  + std::map: Key-value pairs, unique sorted keys, O(logN) access.
* **RTTI (Run-Time Type Information):** Mechanism to determine object type at runtime.
  + **dynamic\_cast:** Safe downcasting for polymorphic objects (returns nullptr or throws bad\_cast).
  + **typeid:** Returns std::type\_info object with type name (.name()). Works on polymorphic objects to give dynamic type; otherwise, static type. Requires <typeinfo>.

Which header file must you include to use the typeid operator? a) <iostream> b) <vector> c) <typeinfo> d) <memory>

===============================Imp mcq I wrong ===========================

**Question 3: Select the right syntax of a C++ function of a class from the listings below:**

**a) virtual void getArray(int size, int\* array);**

b) virtual void getArray(int size, int[] array);

c) virtual void getArray(int size, []int array);

d) virtual int[] getArray(int size);

🡺 **\r (Carriage Return):** Moves the cursor to the beginning of the current line without advancing to the next line. This can result in overwriting the existing content on that line.

🡺**\n (Newline/Line Feed):** Moves the cursor to the beginning of the *next* line. This is the most common way to advance to a new line in C++ output.

🡺 cin: This is the standard input stream object itself, not an operator8

**Question 12: Which feature of OOP allows to have a objects of one class in an another class?** a) Inheritance b) Abstraction c) **Containment** d) Modularity.

**c) Containment (also known as Composition or "has-a" relationship):** This is the OOP feature where one class includes an object of another class as a member. For example, a Car class might "contain" an Engine object or Wheel objects. This allows you to build complex objects from simpler ones.

class Engine { /\* ... \*/ };

class Car {

Engine carEngine; // Car 'has-a' Engine (Containment/Composition)

// ...

};

Question 19: Which operator has highest precedence in \* / %? a) \* b) / c) % **d) All have same precedence**

**🡺** Static data members functions can access **only** static data members" is FALSE because it omits the fact that they can also access static member functions.==> **Static functions do not support polymorphism**: **TRUE**

🡺 few operators that **cannot** be overloaded. These are:

* **.** (Member selection operator)
* **.\*** (Member pointer selection operator)
* **::** (Scope resolution operator)
* **?:** (Ternary conditional operator)
* **sizeof** (Sizeof operator)
* **typeid** (Type information operator)
* **static\_cast**, dynamic\_cast, reinterpret\_cast, const\_cast (Casting operators)
* **Question 28: Which of the following permits function overloading on C++?** a) type b) number of arguments c) type & number of arguments d) number of objects
* The correct option is **c) type & number of arguments**

28. Which function is commonly used to sort an array in C++?

A) sort(arr);

B) sort(arr, arr+n);🡺 In C++, sort() from <algorithm> is used as sort(start\_pointer, end\_pointer);.

C) array.sort();

D) arr->sort();

🡺 Referenence args are void pointers.

**Question 33: Which header file is used to manipulate the string?** a) iostream b) iomanip(I/O manipulators) c) string d) container

The correct option is **c) string**

**Question 38: What is the best way to convert int to string according to C++ standard?** b) std::string s = std::to\_string(42);

🡺  A **Vtable (Virtual Table)** is a mechanism used by C++ compilers to support **runtime polymorphism**.

 A class will have a Vtable (and each object of that class will have a Vptr, a pointer to that Vtable) if and only if **it declares or inherits at least one virtual function**.

**Question 11: Which of the following is the correct way of declaring const in C++?** a) const b) #define c) enum d**) All the Above**

 **B. Persistent Object:** This term specifically refers to an object whose state and data survive beyond the lifetime of the program that created it. This typically involves storing the object's state in a non-volatile medium like a file, database, or network.

 **A. Global Object:** A global object lives for the entire duration of the program's execution, but its existence ends when the program terminates. It does not persist *between* executions.

 **C. Genericity:** This is an OOP concept (like templates in C++) that allows you to write code that works with different data types without being rewritten for each type. It's unrelated to object lifetime.

 **D. Delegation:** This is an OOP design pattern where an object, instead of performing an operation itself, passes (delegates) the operation to another object. It's unrelated to object lifetime.

Object-Based Programming Language supports A. Inheritance B. Polymorphism C. Encapsulation D. All of the above **Object-Based Programming Languages** support **objects and encapsulation**, but **do not support inheritance or polymorphism**.

🡺 Abstraction is crucial to understanding **Object**

Object oriented design decomposes a system into A. Classes B. Objects C. Structures D. Methods

The correct option is **B. Objects**.( While objects are instances of classes,)

If a class member function is declared a const, the function🡺 Does not change the value of any data member of implied object

What is a class? A. It is a region of storage. B. **It defines a data type**. C. It is exactly same as a struct in c. D. All of the above.

🡺 The protected access specifier is specifically designed for use in **inheritance hierarchies**. Members declared protected are accessible within the class that declares them and by its **derived classes**

**🡺** **he virtual keyword in C++ is precisely what enables runtime polymorphism (also known as dynamic dispatch or late binding).**

🡺Pointer to abstract class can be created & Reference to abstract class can be created

🡺 What does an empty class contain? A. Default constructor B. Copy constructor C. Address of operator D. All of the above

The correct option is **D. All of the above**.

In protected derivation

A. P**rotected and public members of base class become protected**

B. Private, protected and public members of base class become protected

C. Private, protected and public members of base class become private

D. Protected and public members of base class become private

🡺 **Class istream : public virtual ios**.

🡺 Class istream in iostream.h is defined as A. Class istream : public ios B. Class istream : public virtual ios C. Class istream : public iostream D. Class istream : public virtual iostream

The correct option is **B. Class istream : public virtual ios**.

🡺 What is the size of empty class? A. 0 bytes B. 2 bytes C. 1 byte D. 4 bytes

The correct option is **C. 1 byte**.

🡺 std::cout << (std::cout<<" Hello ") << " world "; output 🡺 Hello some\_address\_value world

A file can be tied to your program by defining an instance of

A. fstream

B. ifstream

C. ofstream

D. **All of the above**

**The size of the object is the sum of size of all non-static data**.

🡺

🡺

🡺

🡺

🡺

🡺

🡺

🡺

🡺

🡺

🡺