Q/A for-Inheritance Topics: -

1) If both base class and derived class having a function with same, and when we are calling that function in derived which Function will be called?

Ans) It will call the derived class function.

2) If we want to call base class function in derived class which is also having function with same, then how we will call that function?

Ans) You can call the base class function from the derived class using the scope resolution operator (: :)

Base:: Function Name ();

3) What exactly happens when we add virtual in diamond problem?

Ans) When we add virtual inheritance in the diamond problem, it ensures that only one instance of the common base class is shared between the derived classes, preventing ambiguity. This avoids multiple constructions of the base class and resolves the issue of ambiguity in method calls and member access.

4)What is the need of inheritance?

Ans) Inheritance is needed to promote code reusability and establish a hierarchical relationship between classes.

5)What will happen in private inheritance? It will call the derived class function.

Ans) In private inheritance, the public and protected members of the base class become private in the derived class. This means they are not accessible from outside the derived class, but the derived class can still use them internally.

6)Base class having public member function, and a class is derived using private inheritance, can we access base class member function?

Ans) When a base class has a public member function and a derived class inherits privately, the derived class can still access the base class's public member functions internally. However, outside the derived class, the public member functions of the base class are not accessible because of the private inheritance.

7) Why are Destructors called in reverse order?

Ans) Destructors are called in reverse order because the derived class depends on the base class for initialization, and the derived class resources should be cleaned up first. This ensures that the base class resources are properly freed after the derived class's resources are cleaned, preventing any potential access to base class members after they are destroyed.

8) Parametrized constructor in base, creating a derived object, how the changes for this implementation?

Ans) When a parameterized constructor is used in the base class, and you create an object of the derived class, the derived class constructor must explicitly call the base class constructor to pass the required parameters. The derived class constructor needs to call the base class constructor with the necessary arguments using the initializer list. If not explicitly called, the compiler will attempt to call the default constructor of the base class (if it exists). If the base class has no default constructor, it will result in a compilation error.

9) Multiple Inheritance, both base classes having function with same name, then which function called?

Ans) In the case of multiple inheritance, if both base classes have a function with the same name, the compiler will not know which function to call unless you specify which base class function you want to call using the scope resolution operator.

10) What is default access specifier for inheritance?

Ans) The Default access specifier for inheritance is “Private”.

11) Can we do inheritance in structures?

Ans) Yes, inheritance is possible in structures in C++.

12)What is the main difference between structure and class?

Ans) The Main difference between structure and class lies in their default access specifiers.

13)Can we inherit structure from class?

Ans) Yes, We Can inherit structure from a class vice versa class from a structure.

14) How to access private members from a class in inheritance?

Ans) A) In C++, private members of a class are not accessible directly by any class, including derived classes, because they are encapsulated for data protection. However, there are several ways to access private members of a base class from a derived class.

Here are some common methods to access private members:

1.Using Public/Protected Getter and Setter Methods:

2.Using Friend Function or Friend Class.

Q/A for Polymorphism Topics :-

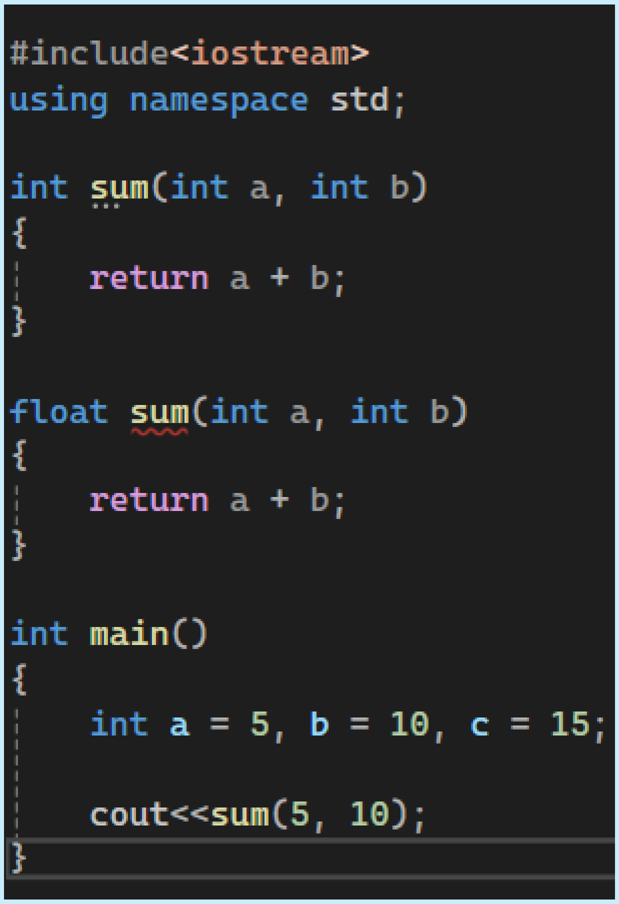
Q1) What is the behavior of function if two different functions return type is different but parameter is same?

Ans) We cannot overload the functions based on the return type alone, return type cannot be taken for the function overloading

--It should be either number of arguments to the function or type of parameter .These things should be differ then function overloading is happen

--If function parameter is same and only return type is different show in this case it will ambiguity so it will throw and compilation error

Eg:-



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

int sum(int a, int b)
{
    return a + b;
}

float sum(int a, int b)
{
    return a + b;
}

int main()
{
    int a = 5, b = 10, c = 15;

    cout<<sum(5, 10);
}
```

We cannot overload using return type alone, this will give error

Q2) How will the compiler know which function should call in function overloading?

Ans) with name mangling compiler will know .

Q3)How compiler will call internally c7+=c2 ?

Ans) c7.operator+=(c2)

Q4) In pre Increment why we are using '&' and Post Increment function why we are not using '&'

Ans) Scope in this function only

Pre-Increment (++x):

Efficient for chaining as it avoids unnecessary copying.

Returns a reference to the incremented value.

Post-Increment (x++):

Requires a copy because it must return the original value before the increment.

Returns by value, not by reference.

Q 5) why virtual function size is 4/8?

Ans) Because of virtual pointer, because pointer takes 4 bytes in 32-bit, 8 bytes in 64-bit platform

Q 6) Can we overload the function with default value and without default value and everything is same?

Ans) No, we cannot overload a function in C++ if the only difference between the functions is the presence or absence of default arguments. This is because default arguments are resolved at compile time, leading to ambiguity.

Q7) Function overloading with member function const and non-const (v.v.i)

Ans) Default argument is not required for overriding

--In C++, we can overload member functions based on whether they are const or non-const. This distinction is made to handle situations where a member function can modify the object or not. The const qualifier indicates that the function will not modify the object (i.e., it is read-only), while a non-const member function can modify the object.

Q8) How memory leak will happen if we having references

Ans) In C++, a memory leak occurs when dynamically allocated memory (e.g., using new or malloc) is not properly deallocated (e.g., using delete or free). If references to the allocated memory persist but are no longer used or accessible, the memory cannot be reclaimed, causing a leak. This happens because references do not own memory and cannot trigger automatic cleanup.

Q9)If in default parameter how compiler will be checking value

Ans) Checking value from right to left

Q.10)Why do we need const member ,without constant we can access or not?

Ans) No, we can access, the const function with non const object and we cannot access non- const function with constant object

Q.11)Suppose in a function two argument is there then Can we pass first argument as default ?

Ans) No we can't give default value as left side because Default parameter is always in Right side

--Calling conversion

Q.12) Best practise for destructor resource deletion?

Ans) Adding if condition is good practise for resource deallocation

```
if(data!=nullptr)

{

Delete data;

data=nullptr;

}
```

Q.13) Why Virtual Constructor will not possible?

Ans) Because constructor itself created a virtual tables and it will assigns the virtual pointer.

This will used in virtual function.

Q.14) Pure virtual destructor is possible or not?

Ans) Yes, it can be possible in C++11 but there is no meaning of this pure Virtual destructor .

Eg:-virtual ~Base() =0;

Q.15) Difference between NULL and nullptr ?

Ans) nullptr was used to avoid the ambiguity

null is always treats as integer value 0 or void* 0, null is macros

Eg)

nullptr is keywords

```
void fun1(int x)

{

cout<<"fun1";

}
```

```
void fun1(int *c)

{

cout<<"fun2";

}
```

when I call NULL then it will call fun1 and when I will call nullptr then it will call the char ptr func

Q.16) What if I call virtual function within a Constructor ?

-->Base Class Version Called: If a virtual function is called within a constructor, only the version defined in the constructor's class is invoked, even if the object being created is of a derived type.

Reason: During the constructor's execution, the vtable is not fully initialized, so the virtual dispatch mechanism does not apply.

Q.17) What is runtime polymorphism ,function overriding is runtime or compile time polymorphism?

Ans) Function overriding is compile polymorphism because here is not virtual function.

Everything comes under compiler time polymorphism except the virtual things.

Q/A for Pointer Topics :-

Q1) Can we use pointers as reference?

Ans) Yes, we can use pointers in conjunction with references . Here are the examples to show how they can be implemented:

Reference to a Pointer

```
int value = 10;

int* ptr = &value; // Pointer to an integer

int*& refPtr = ptr; // Reference to the pointer
```

Pointer to a Reference

```
int value = 10;

int& ref = value; // Reference to an integer
```

```
int* ptr = &ref; // Pointer to the variable that ref refers to
```

Q2. Why do developer prefer reference over pointers?

Ans) Developers prefer references over pointers due to following reasons:

- Simplicity and readability.
- Safety (No null references, Less error prone).
- Automatic Dereferencing.
- Performance.

Q3. Can we have a method of reference to a pointer?

Ans) Yes, we can have a method of reference to a pointer. Here this function is accepting a pointer argument and inside the function it will modify it. Whatever changes done inside the function will reflect in main also.

```
void modifyPointer(int*& ptr){
ptr = new int(33);
}
```

Q4. Example to remove the constness (Through const_casting)?

```
Ans) void RemoveConstness() {
 int x = 10;
 const int y = 100;
 int* const ptr= const_cast<int*> (&y); // Removing constness of y
 *ptr = 200;
(*ptr)++;
(*ptr)--;
}
```

Q5. Example to adding the constness (Through const_casting)?

Ans) void AddConstness() {

```
   int x = 10;

   int* ptr = &x;


   // Adding constness to ptr

   const int* constPtr = const_cast<const int*>(ptr);


   // Now constPtr is a pointer to a const int, so we can't modify the value of x through
constPtr

   // *constPtr = 20; // This line would cause a compilation error


   // However, we can still modify x through the original pointer

   *ptr = 20;

}
```

Q & A for <u>Constructors and Destructors:</u>

Q1) What if you write the cout in the destructor before the null pointer check?

Ans-> In C++, destructors are called when an object goes out of scope, or explicitly deleted (for dynamically allocated objects). If you write cout in the destructor before performing a null pointer check, you may try to access or output a member or a pointer that could be in an invalid state (for example, after memory is deallocated). This could lead to undefined behavior or accessing uninitialized memory, which can cause crashes or unpredictable results.

Q2) How is constructor initialization list more efficient?

Ans-> A constructor initialization list initializes class members directly when an object is created. This is often more efficient than assigning values in the constructor body for several reasons:

Avoids default construction: Members are initialized directly with the values in the initialization list, avoiding any default construction followed by assignment.

For const members: If a class has const members, they must be initialized via the initialization list because they cannot be assigned in the body of the constructor.

For reference members: Reference members also must be initialized in the initialization list.

Q3) What if you remove & in the copy constructor?

Ans-> The & in the copy constructor signifies pass-by-reference, meaning you pass the object without copying it, which avoids unnecessary duplication. If you remove the &, you'll pass the object by value, which means a temporary copy of the object will be created, potentially leading to:

Unnecessary overhead: It will call the copy constructor itself, leading to recursion if not handled carefully.

Performance issues: Creating an unnecessary copy of the object can significantly affect performance, especially for large objects.

If we remove the reference (&), the copy constructor will receive the object by value, which will cause an issue since it will try to copy itself recursively.

Q4) In what cases is the move constructor called?

Ans-> The move constructor is called when an object is being moved (instead of copied). The move constructor transfers ownership of resources from one object to another, leaving the source object in a valid but unspecified state.

The move constructor is invoked in the following situations:

When an rvalue (temporary object) is passed to the constructor.

When a temporary object is returned from a function and used to initialize another object.

When you explicitly use std::move to convert an lvalue to an rvalue.

Q5) When does the copy constructor get called?

Ans-> The copy constructor is called in the following situations:

When passing an object by value (either as a function argument or return value).

When an object is explicitly copied, such as through initialization or assignment (i.e., a new object is created as a copy of an existing one).

When returning an object by value from a function (unless Return Value Optimization (RVO) or Named Return Value Optimization (NRVO) is applied).

The copy constructor creates a new object as a copy of an existing one. It is typically called when an object is passed by value or assigned to another object.

Q6) How can you return from a constructor? The exceptions and all ?

Ans-> In C++, a constructor cannot explicitly return a value. Constructors are special functions that are responsible for initializing an object and are called automatically when an object is created.

However, constructors can throw exceptions if something goes wrong during the initialization of the object. If a constructor throws an exception, the object is not fully constructed, and any memory or resources allocated by the constructor (e.g., dynamic memory) need to be cleaned up before the exception propagates.

If an exception is thrown in a constructor:

The object is considered not to have been constructed.

The destructor is not called (since the object does not exist in a valid state).

If resources were allocated before the exception, they need to be cleaned up manually or through RAII (Resource Acquisition Is Initialization).

Exception safety: If you have resources to manage (like dynamic memory), ensure that exceptions are handled properly to avoid resource leaks. Using smart pointers (like std::unique_ptr) can automate resource management.

noexcept: You can declare a constructor or function as noexcept if it does not throw any exceptions. This helps the compiler optimize and improve performance.

Q7) What constructor will be called in the statement MyClass myObj[100] and how many times?

Ans-> When you declare an array of objects like this: MyClass myObj[100];, the default constructor of MyClass will be called 100 times, once for each element in the array.

If MyClass has a constructor that doesn't take any arguments (the default constructor), it will be called to initialize each of the 100 elements. If no default constructor is explicitly defined, the compiler will generate one, provided no other constructors are defined, and the object types are trivially default-constructible.

If the class has a constructor that takes arguments, and you want to use it, you need to initialize the array with the required arguments, or else you get a compilation error.

Q8) Can you try calling the constructor and destructor of the class explicitly?

Ans-> In general, the constructor is automatically invoked when an object is created. C++ does not provide a direct way to "call" a constructor without creating an object. However, there is a concept called placement new that allows you to construct an object in pre-allocated memory. The constructor in this case is explicitly called at that memory location, but this is done as part of the object creation process and is not something done manually like calling a function.

Key Point: You cannot directly call a constructor like a regular function. The constructor is implicitly called when an object is instantiated. The only exception to this is using placement new, where you explicitly create an object in a pre-allocated memory space.

DESTRUCTOR

While the destructor is generally called automatically when an object is destroyed, there are cases when you might want to call the destructor manually:

For objects created with placement new:

When you use placement new to allocate memory and construct an object in that memory, you have to manually call the destructor before deallocating the memory. This is because delete will not automatically call the destructor for objects created via placement new.

When the destructor is non-trivial (complex cleanup):

In cases where the destructor needs to be invoked explicitly to handle cleanup (e.g., releasing resources), it can be done, but only in very specific situations, usually for objects created with placement new or when working with custom memory management.

Key Point: Just like constructors, destructors are typically called automatically when an object is destroyed. However, when you use placement new, you may need to explicitly call the destructor to ensure proper cleanup.

Q & A for Memory management:

Dangling Pointer:

When a pointer is pointing at the memory address of a variable but after some time that variable is deleted from that memory location while the pointer is still pointing to it, then such a pointer is known as a dangling pointer. Think of it like having the address of a house that was demolished - the address still exists in your records, but there's nothing valid there anymore.

Q1). What'll happen if C++ application contains memory leak?

Ans). A memory leak in a C++ application occurs when dynamically allocated memory isn't deallocated, leading to:

1. **Increased Memory Usage**: The application consumes more memory over time.
2. **Resource Exhaustion**: System memory can run out, causing crashes or allocation failures.
3. **Performance Issues**: Slower performance due to high memory usage and potential swapping.
4. **System Instability**: Can affect overall system performance or crash critical processes.
5. **Debugging Challenges**: Leaks may not be obvious immediately and are hard to trace.

Prevention:

- Use smart pointers (std::unique_ptr, std::shared_ptr).
- Utilize tools like Valgrind or static analyzers.
- Follow RAII and conduct thorough code reviews and testing.

i). Memory corruption: When you try to use a dangling pointer to access or modify data, you're essentially working with memory that could now belong to another program or part of your program. This can corrupt data in unpredictable ways.

ii). Program crashes: Accessing a dangling pointer often leads to segmentation faults or access violations, causing your program to crash.

iii). Security vulnerabilities: In some cases, dangling pointers can be exploited by attackers to execute malicious code or access sensitive data.

Memory Limits:

- In **32-bit applications**, the addressable memory is capped at **2GB** per process, so a leak can quickly exhaust available memory, leading to crashes.
- In **64-bit applications**, the larger address space can delay issues, but the system's physical and virtual memory limits will still be impacted over time.

Impact on Other Applications: While memory leaks in one application generally don't directly affect others, excessive swapping and reduced available memory can degrade the performance of the entire system.

Crashing and Segmentation Faults: When memory is exhausted or improperly managed, the application may crash or encounter **segmentation faults**, especially during memory accesses involving swapped memory or corrupted pointers.

Performance Degradation: Excessive memory usage causes the system to rely on **swapping** (moving data between RAM and disk), significantly slowing the application.

No Recovery Without Freeing Memory: Once memory is leaked, the application cannot reclaim it, leading to ever-growing memory consumption.

==

Q2). delete vs delete[] in C++.

Ans). The delete and delete[] operators are used to free dynamically allocated memory, but they differ in how they handle the allocated memory:

delete

- Used to free memory allocated for a **single object** using new.
- Calls the destructor of the object being deleted (if applicable).

- Does not free additional memory beyond the allocated object.
- In the case of polymorphic classes, delete ensures the proper destructor is called based on the vptr (virtual table pointer) if the base class destructor is declared virtual.

delete[]

- Used to free memory allocated for an **array of objects** using new[].
- Ensures that the destructor of each object in the array is called (if applicable).
- Properly releases all the memory allocated for the array.

Using delete on arrays:

- Leads to undefined behavior because it only deallocates the memory for the first element.
- **Using delete[] on single objects**:
- Also undefined behavior, as it attempts to deallocate memory as if it were an array.

The delete and delete[] operators in C++ work differently under the hood due to how memory is allocated and managed for single objects and arrays. Here's a detailed explanation of their workings:

1. Memory Allocation with new and new[]

- **new**: Allocates memory for a single object. The runtime keeps track of the size of the allocated memory.
- **new[]**: Allocates memory for an array of objects. The runtime often stores metadata (such as the number of elements) to ensure proper cleanup during delete[].

2. What Happens During delete

When you call delete ptr;:

1. **Destructor Invocation**:

a. If the object has a destructor, delete calls it to perform any necessary cleanup (e.g., releasing resources).

2. **Memory Deallocation**:

a. After calling the destructor, the runtime calls the memory allocator (e.g., free() or a custom allocator) to release the allocated memory.

3. What Happens During delete[]

When you call delete[] ptr;:

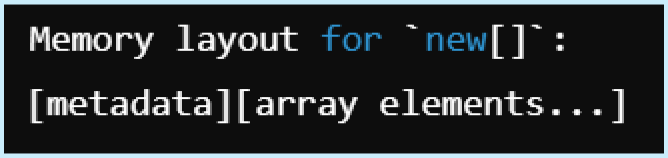
1. **Destructor Invocation for Each Element**:

a. The runtime retrieves the metadata stored during new[] (typically the array size).

b. It iterates through each element in reverse order (to mimic stack unwinding) and calls the destructor for each object in the array.

2. **Memory Deallocation**:

a. Once all destructors are called, the runtime frees the entire memory block allocated for the array.



```
Memory layout for `new[]`:
[metadata][array elements...]
```

Single Object (new): No additional metadata is needed because it's for a single object, and its size is known at allocation time.

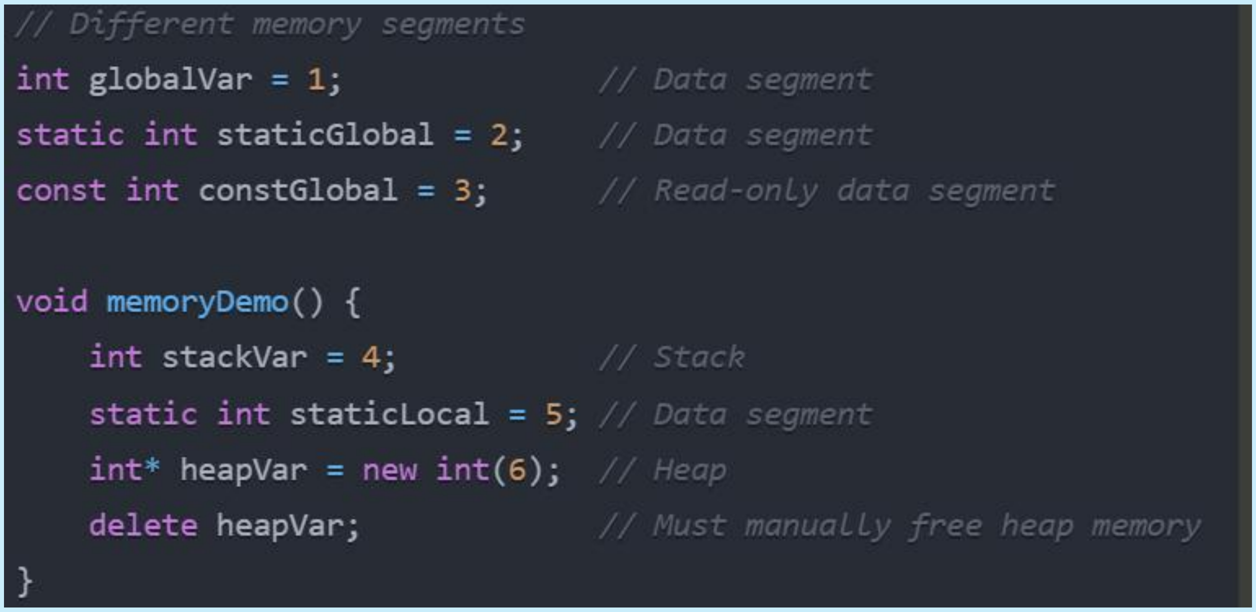
===

Q3). Explain behaviors of local and global static variable.

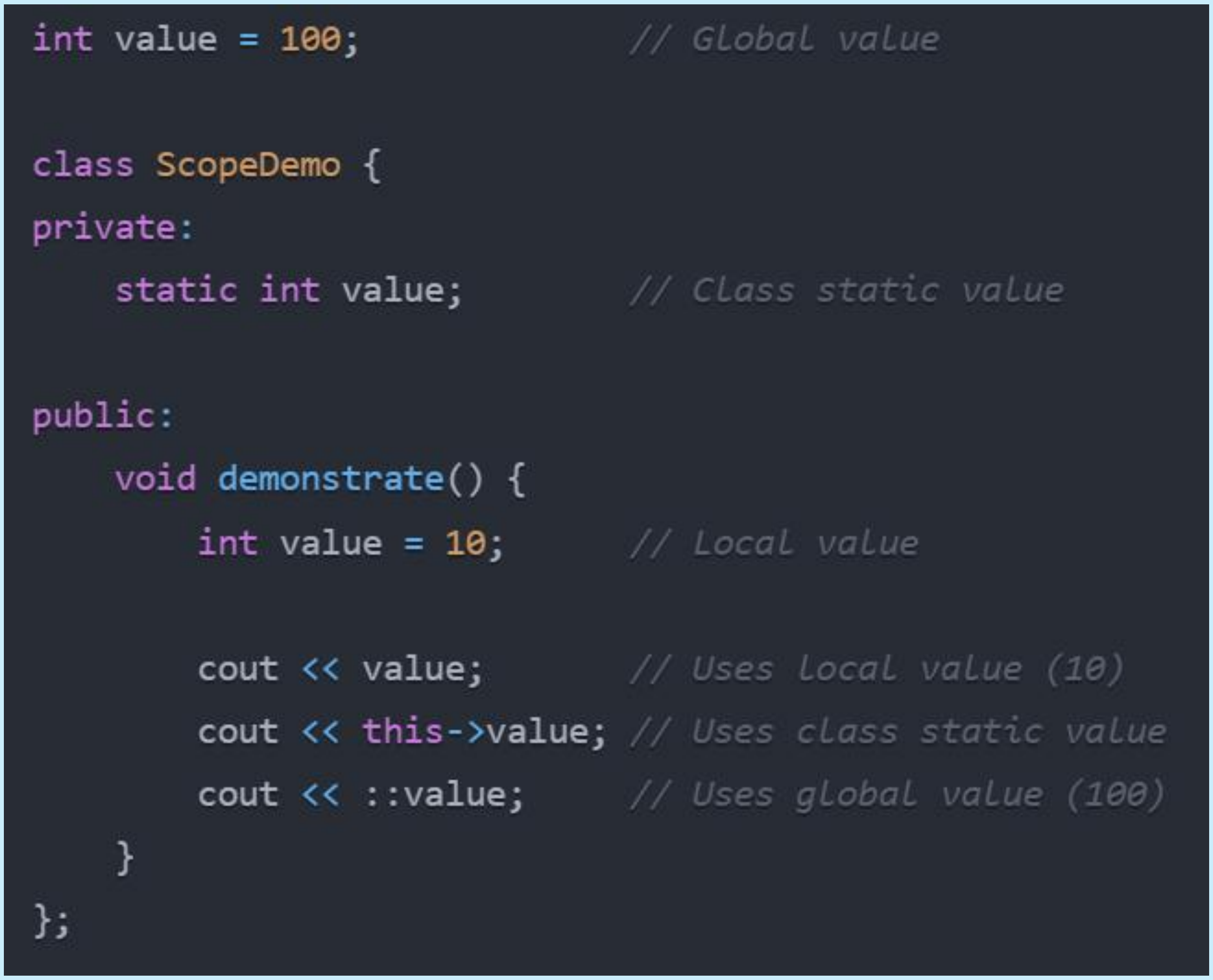
Ans).

- Static and global variables are initialized before main()
- Local static variables are initialized on first function call
- Use static class members for class-wide data
- Be careful with thread safety when modifying static data

- Consider alternatives to global variables when possible

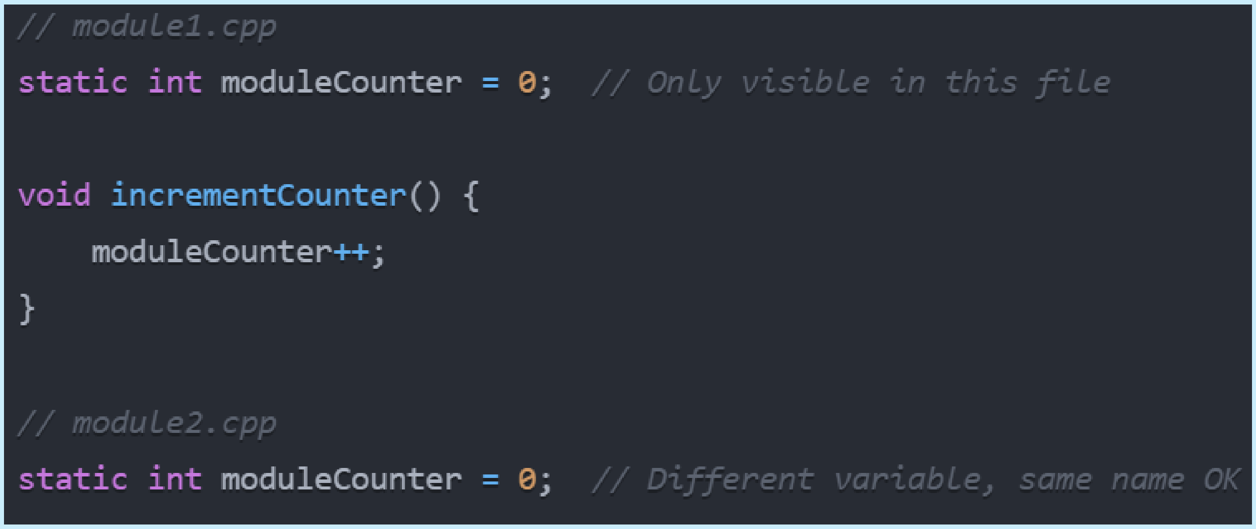


```cpp
// Different memory segments
int globalVar = 1;                // Data segment
static int staticGlobal = 2;      // Data segment
const int constGlobal = 3;        // Read-only data segment

void memoryDemo() {
    int stackVar = 4;             // Stack
    static int staticLocal = 5; // Data segment
    int* heapVar = new int(6);  // Heap
    delete heapVar;               // Must manually free heap memory
}
```

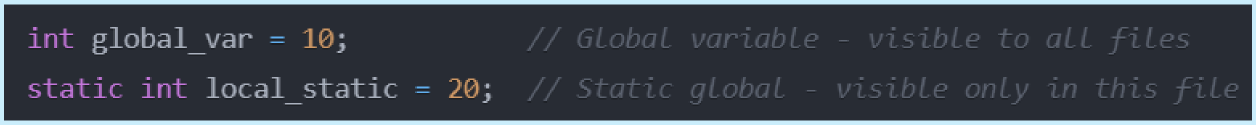


```
int value = 100;             // Global value

class ScopeDemo {
private:
    static int value;        // Class static value

public:
    void demonstrate() {
        int value = 10;      // Local value

        cout << value;       // Uses local value (10)
        cout << this->value; // Uses class static value
        cout << ::value;     // Uses global value (100)
    }
};
```

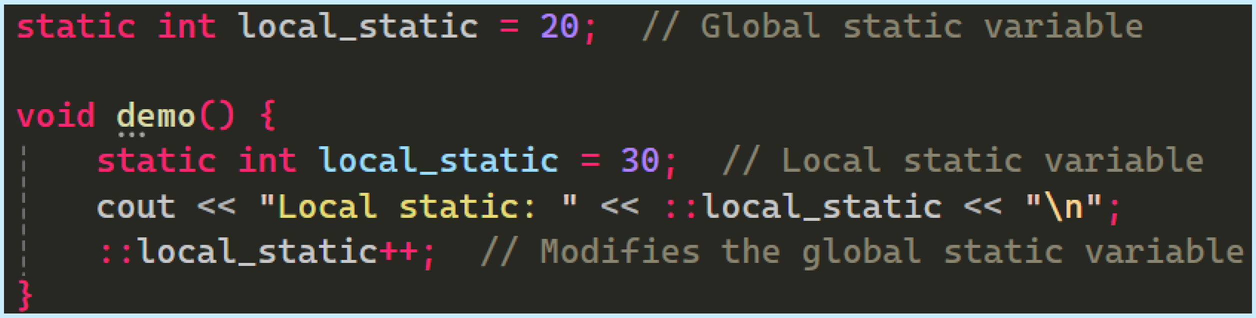
- Static variables inside functions maintain their value between function calls.
- Static variables at file scope are only visible within their translation unit.



```cpp
// module1.cpp
static int moduleCounter = 0;  // Only visible in this file

void incrementCounter() {
    moduleCounter++;
}

// module2.cpp
static int moduleCounter = 0;  // Different variable, same name OK
```

- global_var can be accessed from other files using extern declaration
- local_static is only visible within the current file (translation unit)
- Both exist for the entire program lifetime
- Both are initialized before main() starts
- 

```cpp
int global_var = 10;           // Global variable - visible to all files
static int local_static = 20;  // Static global - visible only in this file
```



```cpp
static int local_static = 20;  // Global static variable

void demo() {
    static int local_static = 30;  // Local static variable
    cout << "Local static: " << ::local_static << "\n";
    ::local_static++;  // Modifies the global static variable
}
```

:: is being used to access the **global scope** variable local_static when there is a naming conflict with a local variable.

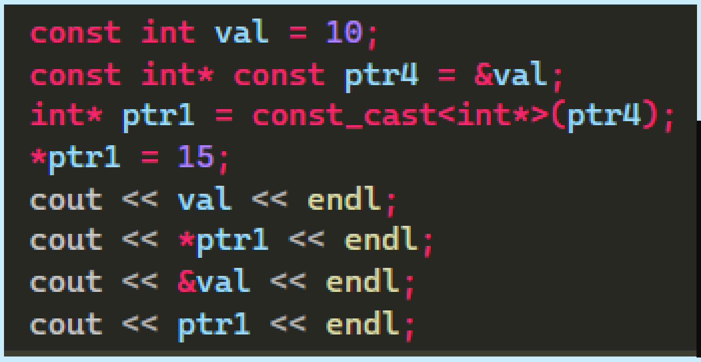
- static int local_static = 30; is declared inside the demo() function.

- This local variable shadows the global variable within the function's scope.
- Without ::, any reference to local_static inside the function would refer to this local variable.

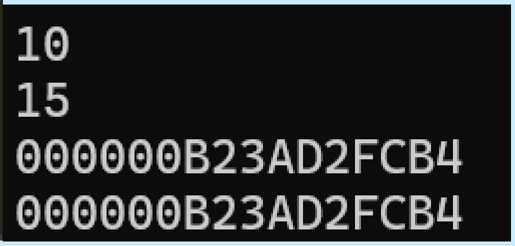
The ::local_static explicitly tells the compiler to use the **global static variable** local_static instead of the local one.

==
====

Q4). In case of const_cast, what's happening in these two scenarios:

A). 

```
const int val = 10;
const int* const ptr4 = &val;
int* ptr1 = const_cast<int*>(ptr4);
*ptr1 = 15;
cout << val << endl;
cout << *ptr1 << endl;
cout << &val << endl;
cout << ptr1 << endl;
```

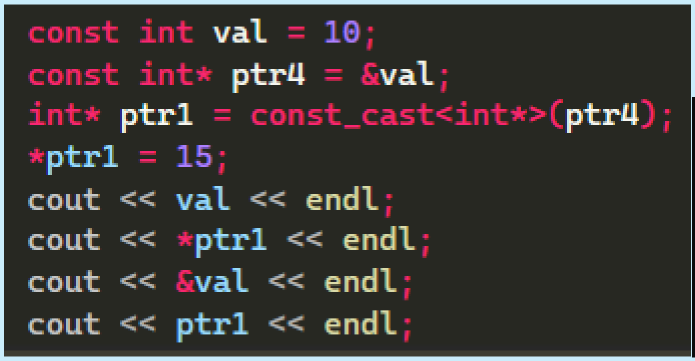


```
10
15
000000B23AD2FCB4
000000B23AD2FCB4
```

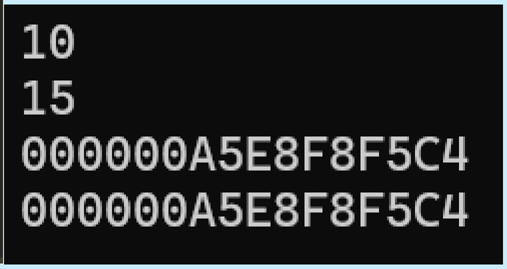
Ans). const_cast removes the const qualifier from ptr4, creating a non-const pointer ptr1. However, modifying val through ptr1 invokes **undefined behavior** because val is declared const.

Attempting to modify val through ptr1. This violates const rules, leading to undefined behavior. The result depends on the compiler and memory protection. Some compilers may allow the modification, while others may leave val unchanged or crash the program.

- val remains 10 because it is immutable.
- *ptr1 shows 15 because the value is updated in a way that violates the const qualifier.
- **This behavior is undefined and should not be relied upon.**

B). 

```
const int val = 10;
const int* ptr4 = &val;
int* ptr1 = const_cast<int*>(ptr4);
*ptr1 = 15;
cout << val << endl;
cout << *ptr1 << endl;
cout << &val << endl;
cout << ptr1 << endl;
```



```
10
15
000000A5E8F8F5C4
000000A5E8F8F5C4
```

Ans). ptr4 is a pointer to a constant integer. Unlike Scenario 1, ptr4 is not constant, meaning the pointer can point to a different address, but the value it points to cannot be modified.

const_cast removes the const qualifier from ptr4, creating a non-const pointer ptr1. Modifying val through ptr1 still results in undefined behavior.

Similar to Scenario 1, this attempts to modify val. Since val is constant, this action violates const rules and invokes undefined behavior.

- val remains 10 because it is immutable.
- *ptr1 shows 15 due to undefined behavior caused by modifying a const variable.
- **This behavior is also undefined and should not be relied upon.**

Summary of Behavior in Both Scenarios

1. **Constant Variable:**
2. val is declared const, so any attempt to modify it through a non-const pointer results in **undefined behavior**.
3. **Memory Address:**

The address of val (&val) and the pointer ptr1 will always be the same, as ptr1 points to val.

4. **Compiler Dependency:**

The behavior of modifying a const variable through a const_cast is undefined and depends on compiler optimizations. Some compilers may allow the modification, while others may crash or leave val unchanged.

5. **Good Practice:**

Avoid using const_cast to modify const variables unless absolutely necessary, as it violates const correctness and leads to unpredictable behavior.

==

Q5). When will dynamic_cast fail?

Ans). When using dynamic_cast with references, if the cast is invalid, it throws a std::bad_cast exception instead of returning a nullptr (as it does with pointers).

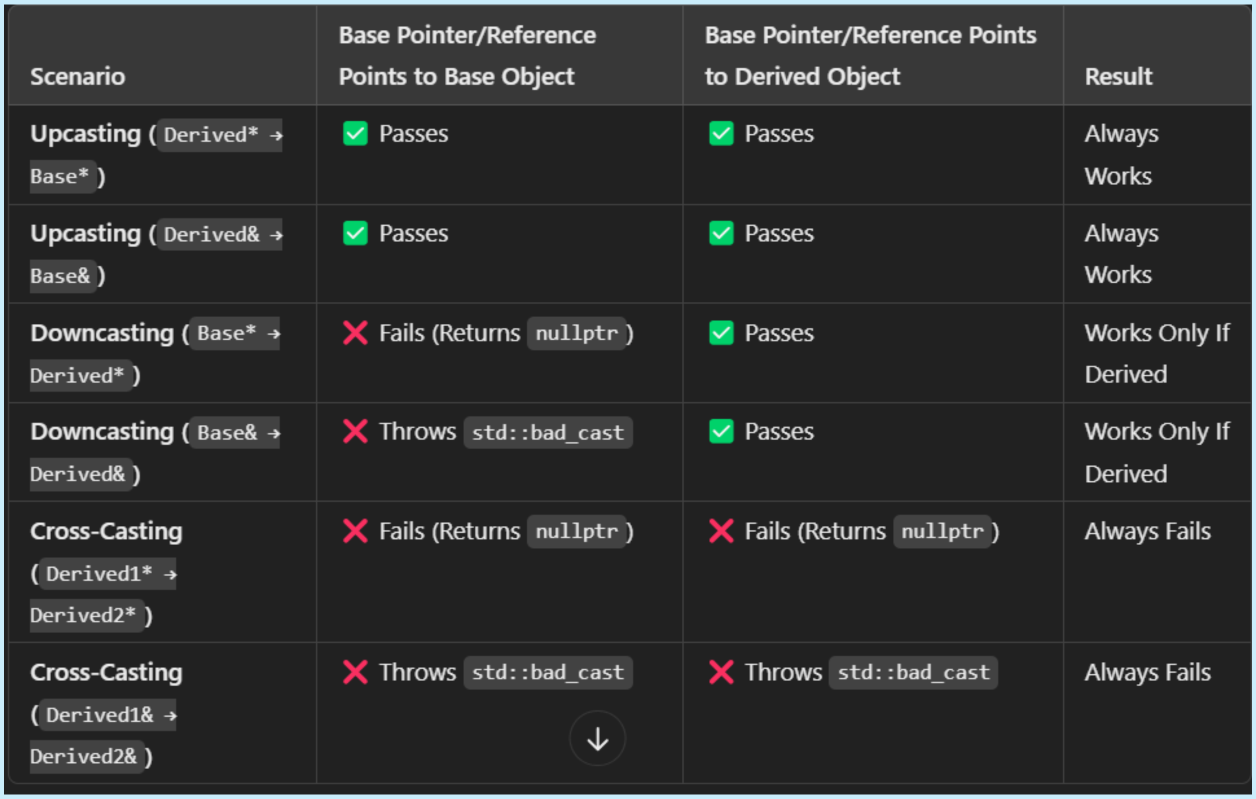
When dynamic_cast Fails

- **Invalid Downcasting**:
  - When trying to downcast a base class pointer/reference to a derived class, but the object is not actually of the derived type.
  - **Pointers**: Returns nullptr.
  - **References**: Throws a std::bad_cast exception.
- **Non-Polymorphic Classes**:
  - Fails at compile time if the base class does not have at least one virtual function.
- **Casting Across Sibling Classes**:
  - Fails when casting between unrelated derived classes, even if they share the same base class.
  - **Example**: Derived1 to Derived2 via a Base*.

==

Q6). Summarize the behavior of dynamic_cast in upcasting and downcasting.

Ans).



Scenario	Base Pointer/Reference Points to Base Object	Base Pointer/Reference Points to Derived Object	Result
Upcasting (`Derived* → Base*`)	✅ Passes	✅ Passes	Always Works
Upcasting (`Derived& → Base&`)	✅ Passes	✅ Passes	Always Works
Downcasting (`Base* → Derived*`)	❌ Fails (Returns `nullptr`)	✅ Passes	Works Only If Derived
Downcasting (`Base& → Derived&`)	❌ Throws `std::bad_cast`	✅ Passes	Works Only If Derived
Cross-Casting (`Derived1* → Derived2*`)	❌ Fails (Returns `nullptr`)	❌ Fails (Returns `nullptr`)	Always Fails
Cross-Casting (`Derived1& → Derived2&`)	❌ Throws `std::bad_cast`	❌ Throws `std::bad_cast`	Always Fails

1. Upcasting (Derived → Base)

- **Safe Operation:**
- Always works because the derived object includes the base part.
  - For pointers: No checks are needed; directly converts.
  - For references: Simply binds to the base part of the derived object.

2. Downcasting (Base → Derived)

- **Successful When Base Pointer/Reference Points to a Derived Object:**

Works if the actual object type is of the derived class.

- For pointers: Returns a valid pointer.
- For references: Succeeds without exception.

- **Fails When Base Pointer/Reference Points to a Base Object:**
  - For pointers: Returns nullptr.
  - For references: Throws std::bad_cast.

3. Cross-Casting (Derived1 → Derived2)

- **Always Fails:**

There's no relationship between the sibling classes.

- For pointers: Returns nullptr.
- For references: Throws std::bad_cast.

==
====

Q7). Mutable and its significance, also how can update the values of mutable variables inside the const member function?

Ans). mutable keyword allows a member variable of a class to be modified even if it is part of an object that is declared as const. Normally, when an object is declared const, all its member variables cannot be modified. However, mutable provides an exception to this rule.

Key Points about mutable:

1. **Allows Modification in const Objects:** A mutable member variable can be updated even in a const object.
2. **Usage in const Member Functions:** const member functions cannot modify non-mutable member variables, but they can modify mutable variables.
3. **Common Use Cases:**

a. For caching values in a class.

b. To track auxiliary data such as logging, debugging, or counters without affecting the object's logical constness.

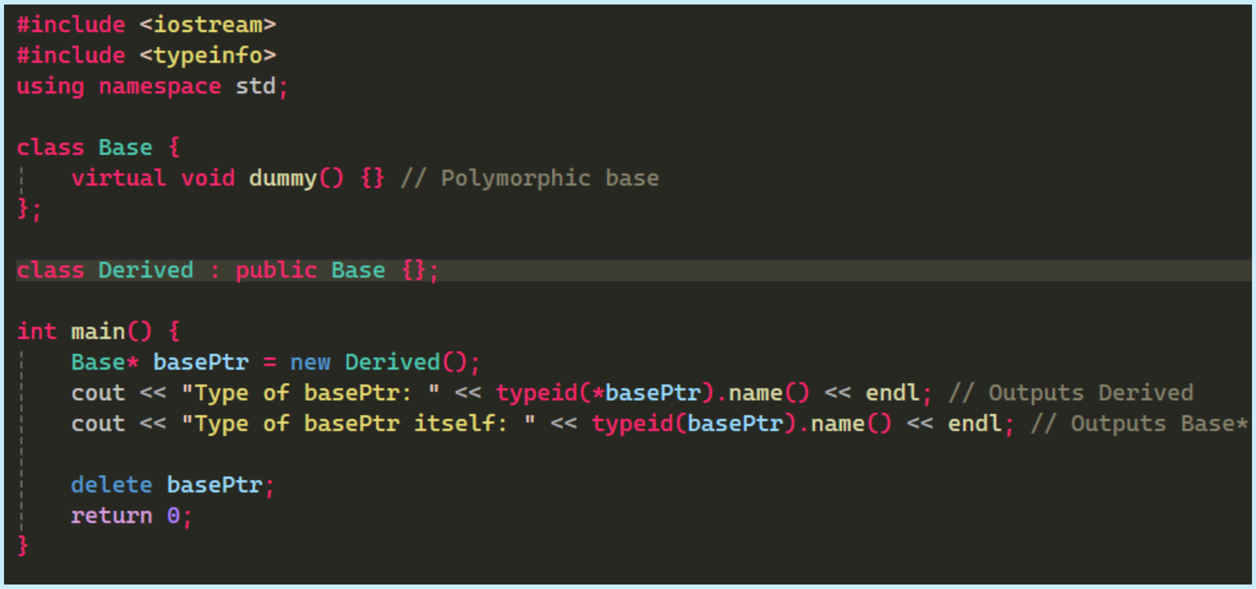
==
====

Q8). How to achieve RTTI (Runtime type identification)?

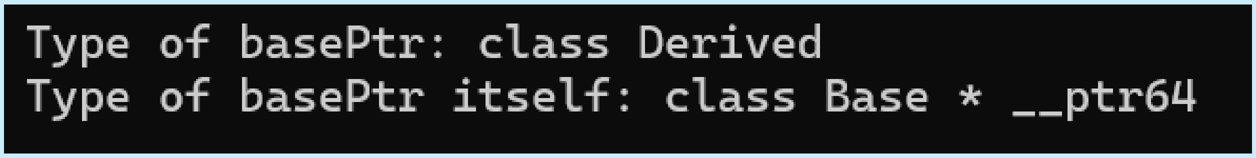
Ans). RTTI is a mechanism in C++ that enables type identification of objects during runtime. It allows a program to determine the **type of an object or pointer** dynamically, especially in cases involving polymorphism.

1. Using typeid Operator:

The typeid operator is used to retrieve type information about objects or pointers. It works with both polymorphic and non-polymorphic types.



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

class Base {
    virtual void dummy() {} // Polymorphic base
};

class Derived : public Base {};

int main() {
    Base* basePtr = new Derived();
    cout << "Type of basePtr: " << typeid(*basePtr).name() << endl; // Outputs Derived
    cout << "Type of basePtr itself: " << typeid(basePtr).name() << endl; // Outputs Base*

    delete basePtr;
    return 0;
}
```

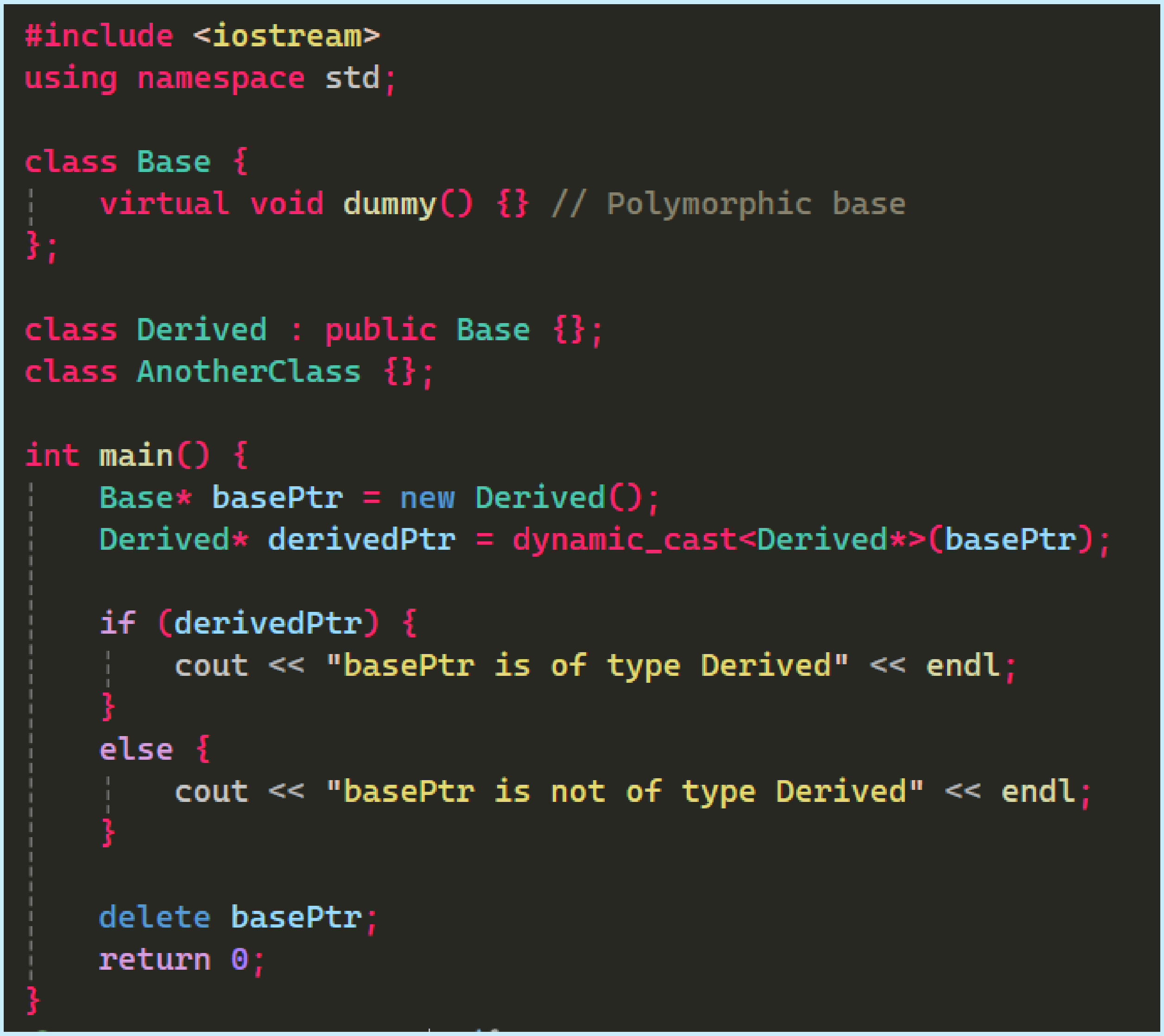


```
Type of basePtr: class Derived
Type of basePtr itself: class Base * __ptr64
```

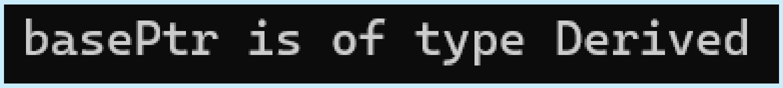
typeid(*basePtr) identifies the actual type (Derived) at runtime.

2. Using dynamic_cast:

dynamic_cast can also help determine the actual type of a polymorphic object by attempting a safe downcast.



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

class Base {
    virtual void dummy() {} // Polymorphic base
};

class Derived : public Base {};
class AnotherClass {};

int main() {
    Base* basePtr = new Derived();
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);

    if (derivedPtr) {
        cout << "basePtr is of type Derived" << endl;
    }
    else {
        cout << "basePtr is not of type Derived" << endl;
    }

    delete basePtr;
    return 0;
}
```



```
basePtr is of type Derived
```

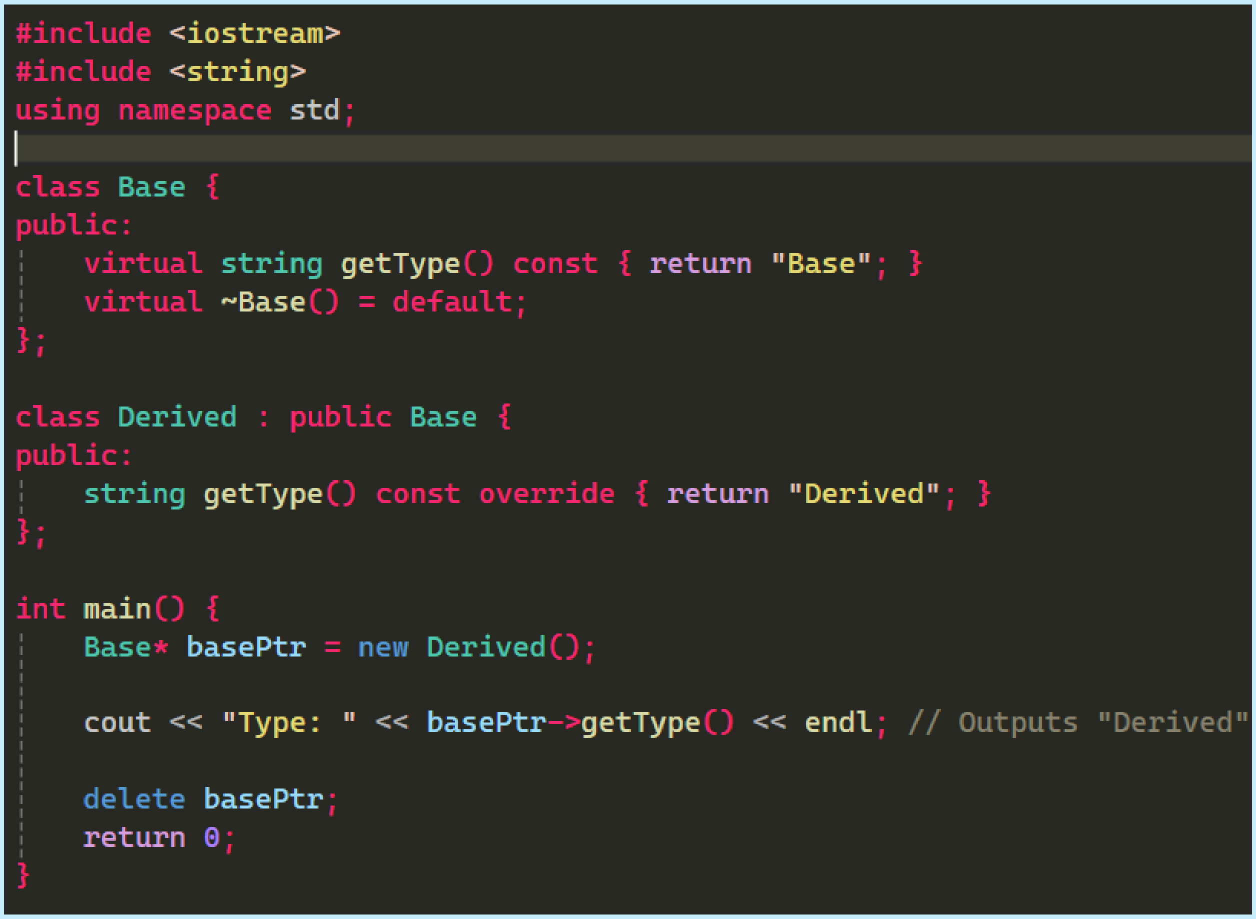
- If the cast succeeds, dynamic_cast returns a valid pointer. Otherwise, it returns nullptr.
- Works only for objects within the same hierarchy.

Achieving RTTI Without Using dynamic_cast:

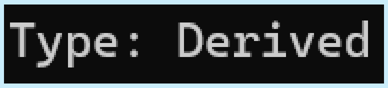
When objects don't belong to the same hierarchy or dynamic_cast can't be used (e.g., non-polymorphic types), RTTI can be simulated manually:

1. Using Virtual Functions (Custom RTTI):

Define a virtual function in the base class that identifies the derived type.



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

class Base {
public:
    virtual string getType() const { return "Base"; }
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    string getType() const override { return "Derived"; }
};

int main() {
    Base* basePtr = new Derived();

    cout << "Type: " << basePtr->getType() << endl; // Outputs "Derived"

    delete basePtr;
    return 0;
}
```



```
Type: Derived
```

==
====

Q9). Does compiler always generate move constructor or on some special conditions?

Ans). The compiler **generates a move constructor** only under specific conditions. These conditions ensure that move semantics can be correctly implemented.

Conditions for Compiler-Generated Move Constructor:

1. **Class Must Not Define Certain Special Member Functions**: If you explicitly define any of the following special member functions, the compiler does **not automatically generate** a move constructor:

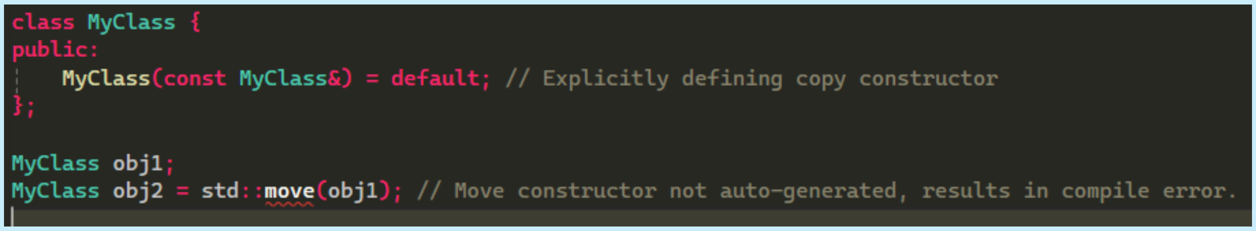
a. Copy constructor

b. Copy assignment operator

c. Move constructor

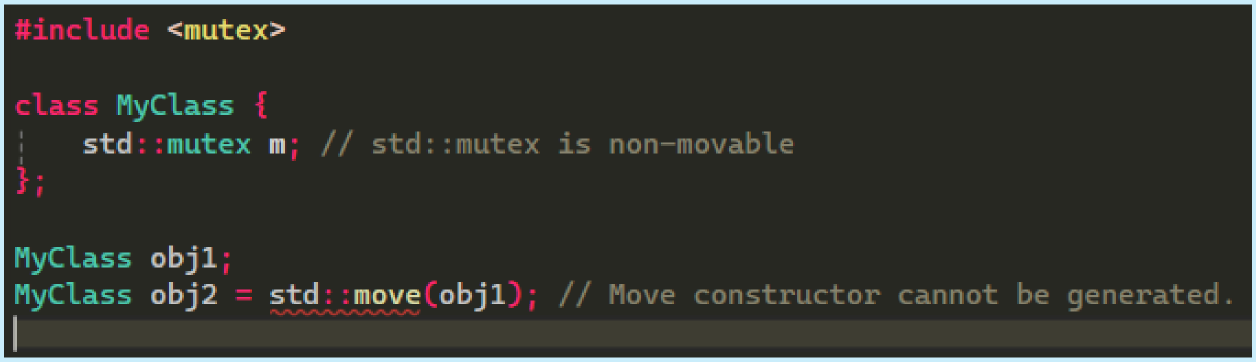
d. Move assignment operator

e. Destructor



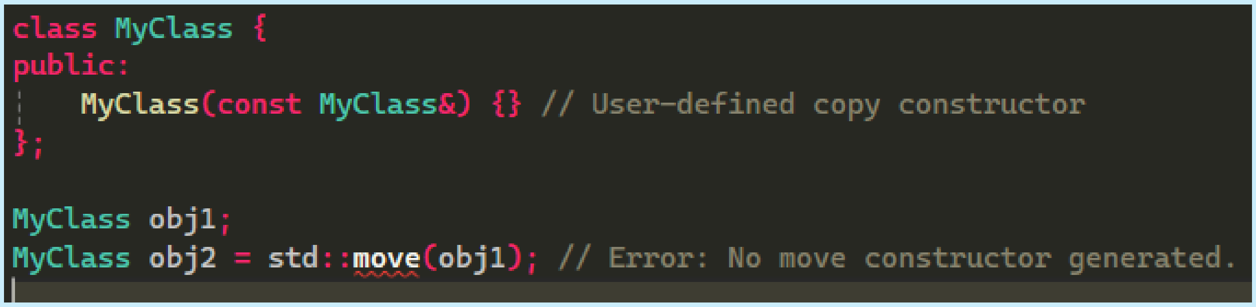
```cpp
class MyClass {
public:
    MyClass(const MyClass&) = default; // Explicitly defining copy constructor
};

MyClass obj1;
MyClass obj2 = std::move(obj1); // Move constructor not auto-generated, results in compile error.
```

2. **Class Must Not Have Non-Movable Members**: If the class contains a member that does not support move semantics (e.g., std::mutex or raw pointers without proper ownership transfer), the compiler **cannot generate** a move constructor.

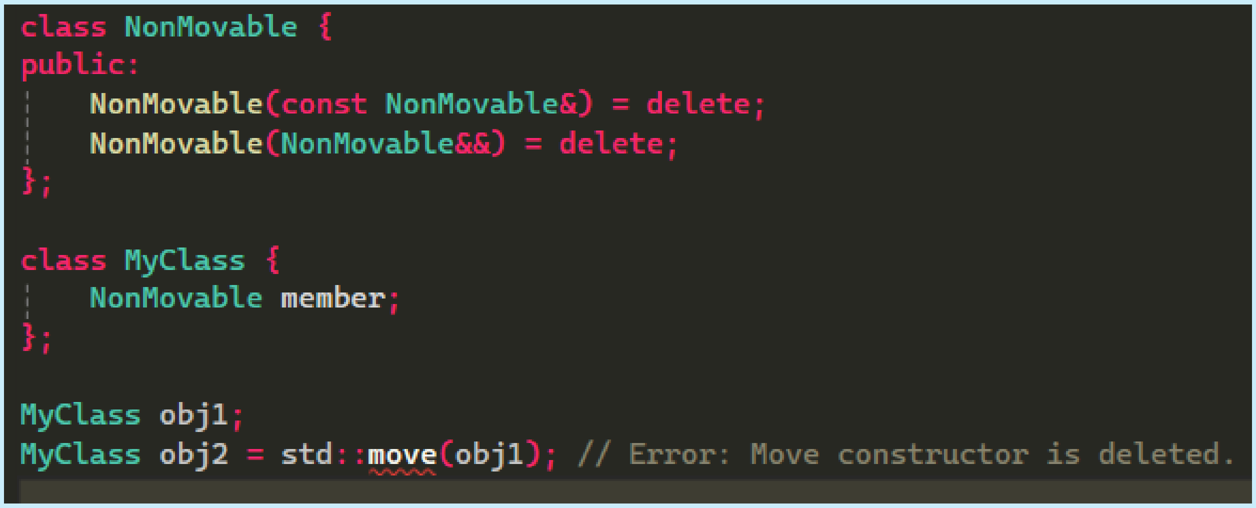


```cpp
#include <mutex>

class MyClass {
    std::mutex m; // std::mutex is non-movable
};

MyClass obj1;
MyClass obj2 = std::move(obj1); // Move constructor cannot be generated.
```

3. **Move Constructor Is Only Generated If No Explicit Copy Constructor Is Defined**: If you define a copy constructor explicitly, the compiler assumes you want control over copying behavior and does not generate a move constructor.


```cpp
class MyClass {
public:
    MyClass(const MyClass&) {} // User-defined copy constructor
};

MyClass obj1;
MyClass obj2 = std::move(obj1); // Error: No move constructor generated.
```

4. **Implicitly Deleted for Certain Members**: If any member of the class has an explicitly deleted or inaccessible move constructor, the class's move constructor will also be implicitly deleted.


```cpp
class NonMovable {
public:
    NonMovable(const NonMovable&) = delete;
    NonMovable(NonMovable&&) = delete;
};

class MyClass {
    NonMovable member;
};

MyClass obj1;
MyClass obj2 = std::move(obj1); // Error: Move constructor is deleted.
```

==
====

Q & A for Vector:

Q.1) In case of Time Complexity and Space Complexity, what to prefer, array or vector ?

Ans)

A.) Memory Storage

Array: Stored in the stack (fixed size, faster access).

Vector: Stored in the heap (dynamic size, slower access due to pointer dereferencing).

B.) Resizing:

Array: Size is fixed at compile-time, no resizing overhead.

Vector: Dynamically resizes, doubling capacity when full, which uses extra memory for future growth.

C.)Memory Usage:

Array: Requires less memory (no extra overhead for resizing).

Vector: Uses more memory due to reserved capacity and dynamic allocation overhead.

D.) When to Prefer:

Use array for fixed-size data (efficient in memory).

Use vector for dynamic-size data (flexible but slightly heavier in memory).

Q.2) Size and capacity relation, why Capacity is getting doubled ?

Ans)The capacity of a vector doubles to optimize performance:

Minimizes Reallocation: Doubling reduces the number of times the vector needs to resize, saving time.

Amortized O(1) Insertion: Frequent reallocations are costly, so doubling ensures insertions remain efficient.

Memory Efficiency: Doubling strikes a balance between using memory and minimizing overhead.

Q.3) Difference between emplace_back vs push_back in vector .

Ans) push_back()

What It Does: Copies or moves an object into the vector.

Usage: You must pass an already constructed object as an argument to push_back().

Behavior:

If the object is not a temporary (rvalue), it will copy the object into the vector.

If the object is a temporary (rvalue), it will move the object into the vector (assuming move semantics are defined for the object).

emplace_back()

What It Does: Constructs the element in place directly in the vector.

Usage: You pass the constructor arguments for the object to emplace_back(), and it constructs the object directly in the vector's storage.

Behavior:

Avoids the need for creating a temporary object.

Eliminates unnecessary copies or moves, as the object is constructed in place.

Q.4) What is Relationship between Size and Capacity ?

Ans) size(): The number of elements currently in the vector.

capacity(): The total number of elements the vector can hold before needing to allocate more memory.

Relationship:

If size() < capacity(): The vector has unused space.

If size() == capacity(): The vector is "full" (current capacity used). Adding more elements will double the capacity to allocate more space.

Q & A for <u>(MAP & SET):</u>

Q1.) Capture return type for methods of map, unordered map, multimap, set, unordered set and multiset.

Q2.) cover the scenario where map is preferred over unordered map.

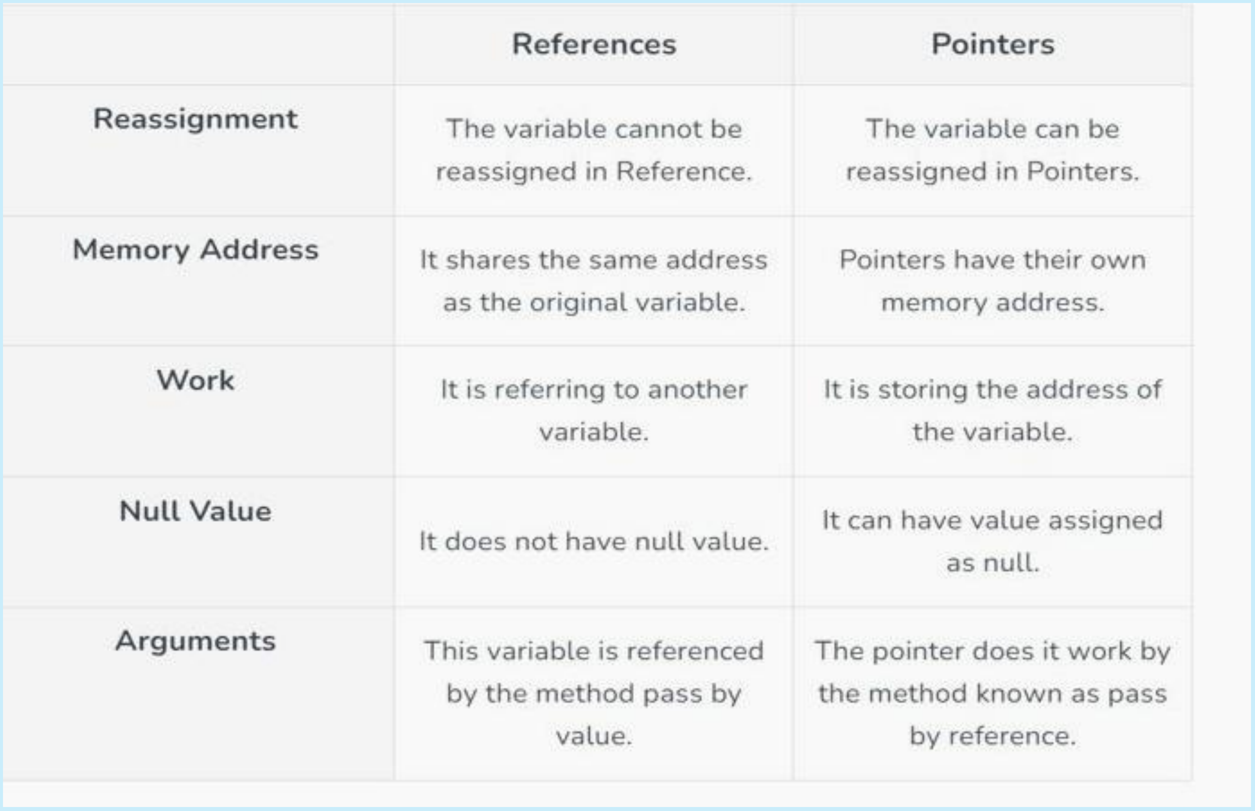
Miscellaneous Questions ->

Q1) Other than Function overloading, operator overloading and function overriding, how can we perform overloading in C++ ?

Ans-> In Generic Template programming

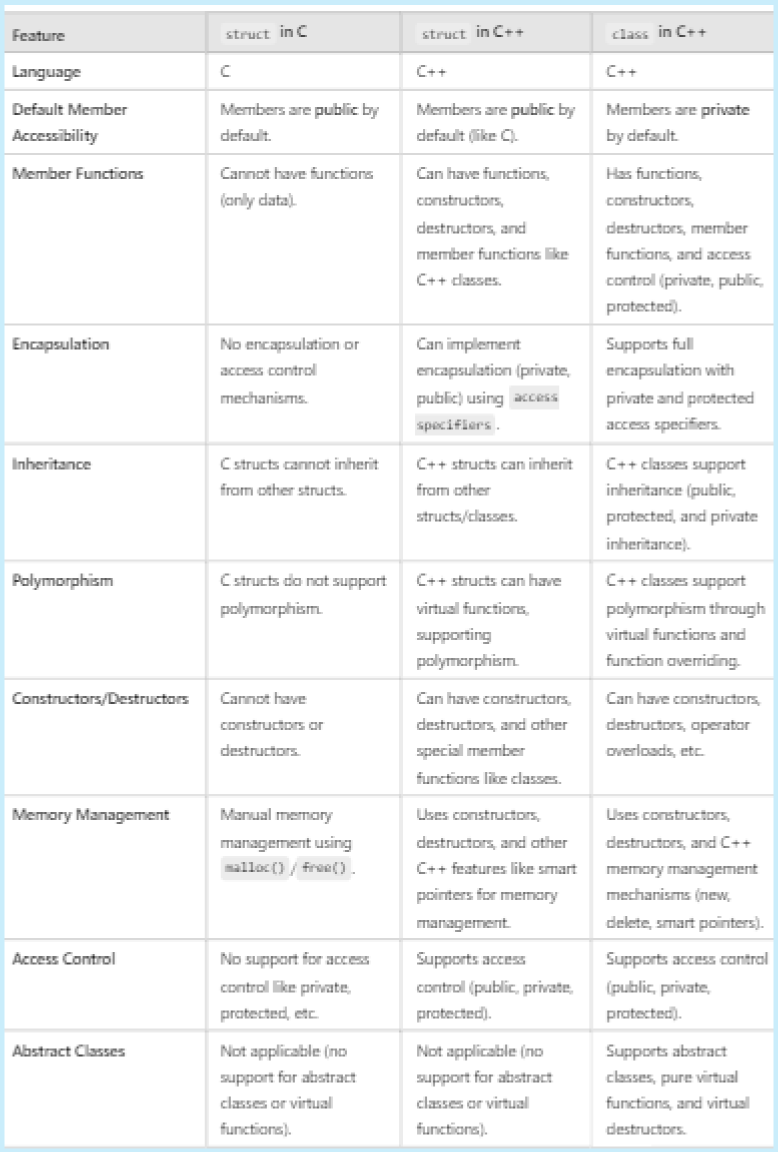
```
#include<iostream>

using namespace std;


template<typename t >

void add(t a, t b) {

   cout << " In template " << endl;

   cout << a << " " << b << endl;

}


int add(int a, int b ) {

   cout << " In function " << endl;

 return a + b;

}


int main() {


    add(3 , 8.0f);

}
```

Q2) Difference between Reference vs Pointers?



	References	Pointers
Reassignment	The variable cannot be reassigned in Reference.	The variable can be reassigned in Pointers.
Memory Address	It shares the same address as the original variable.	Pointers have their own memory address.
Work	It is referring to another variable.	It is storing the address of the variable.
Null Value	It does not have null value.	It can have value assigned as null.
Arguments	This variable is referenced by the method pass by value.	The pointer does it work by the method known as pass by reference.

Q3) What is the difference between Class and Struct in c and cpp?



Feature	`struct` in C	`struct` in C++	`class` in C++
Language	C	C++	C++
Default Member Accessibility	Members are **public** by default.	Members are **public** by default (like C).	Members are **private** by default.
Member Functions	Cannot have functions (only data).	Can have functions, constructors, destructors, and member functions like C++ classes.	Has functions, constructors, destructors, member functions, and access control (private, public, protected).
Encapsulation	No encapsulation or access control mechanisms.	Can implement encapsulation (private, public) using `access specifiers`.	Supports full encapsulation with private and protected access specifiers.
Inheritance	C structs cannot inherit from other structs.	C++ structs can inherit from other structs/classes.	C++ classes support inheritance (public, protected, and private inheritance).
Polymorphism	C structs do not support polymorphism.	C++ structs can have virtual functions, supporting polymorphism.	C++ classes support polymorphism through virtual functions and function overriding.
Constructors/Destructors	Cannot have constructors or destructors.	Can have constructors, destructors, and other special member functions like classes.	Can have constructors, destructors, operator overloads, etc.
Memory Management	Manual memory management using `malloc()` / `free()`.	Uses constructors, destructors, and other C++ features like smart pointers for memory management.	Uses constructors, destructors, and C++ memory management mechanisms (new, delete, smart pointers).
Access Control	No support for access control like private, protected, etc.	Supports access control (public, private, protected).	Supports access control (public, private, protected).
Abstract Classes	Not applicable (no support for abstract classes or virtual functions).	Not applicable (no support for abstract classes or virtual functions).	Supports abstract classes, pure virtual functions, and virtual destructors.

Ans->

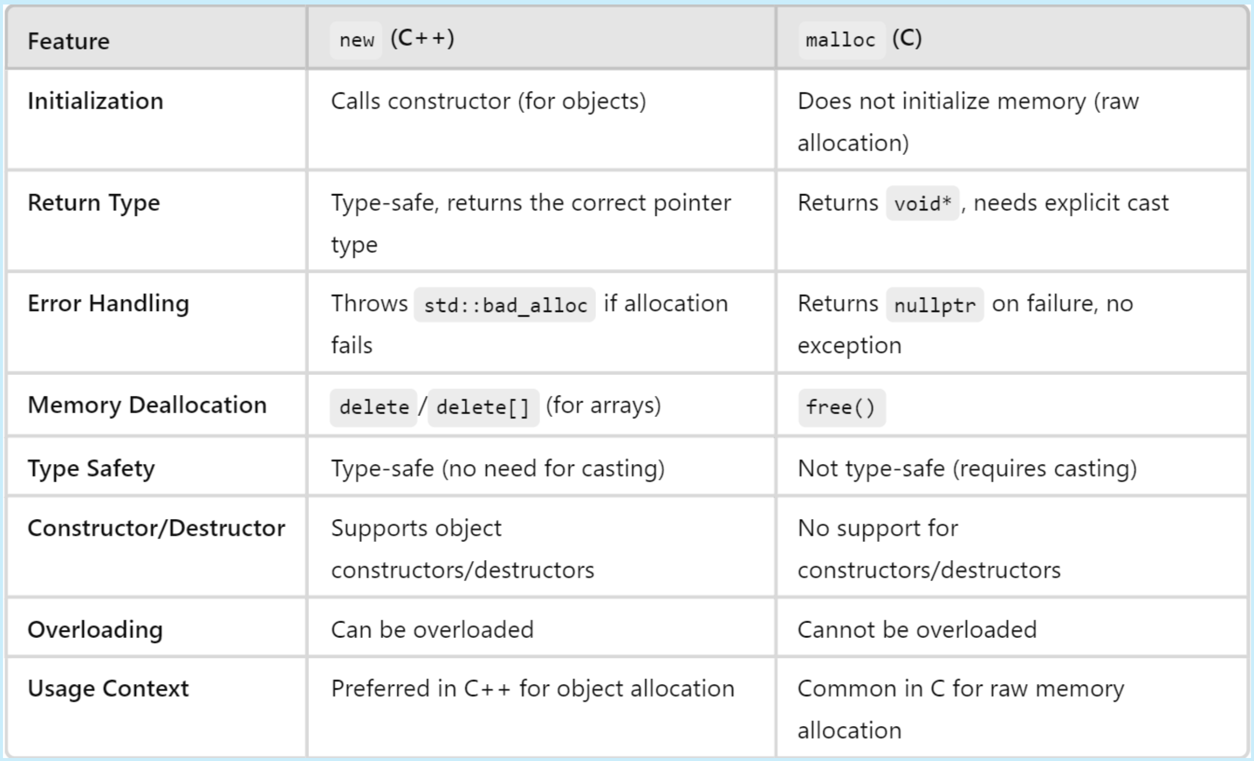
Q4) What is the difference between you allocate obj using new and malloc?

Ans-> New : -new is an operator not a function

- In case of failure returns **bad allocation exception**
- Operator overloading can be done in case of new
- Used for identifying memory leak with CRT functions.
- Destructor will be called.
- Resources will be freed.

Malloc : - it is a function

- In case of failure it returns **nullptr.**
- No resources are freed and no destructor will be called.
- No support for exception handling.
- No memory leaks will be detected.



Feature	`new` (C++)	`malloc` (C)
Initialization	Calls constructor (for objects)	Does not initialize memory (raw allocation)
Return Type	Type-safe, returns the correct pointer type	Returns `void*`, needs explicit cast
Error Handling	Throws `std::bad_alloc` if allocation fails	Returns `nullptr` on failure, no exception
Memory Deallocation	`delete` / `delete[]` (for arrays)	`free()`
Type Safety	Type-safe (no need for casting)	Not type-safe (requires casting)
Constructor/Destructor	Supports object constructors/destructors	No support for constructors/destructors
Overloading	Can be overloaded	Cannot be overloaded
Usage Context	Preferred in C++ for object allocation	Common in C for raw memory allocation

Q5) what will happen in case where you use free with new?

Ans-> Custom Memory Management

- If we call free then destructor won't call

Q6) why there is a need for overloading new?

Ans-> **Debugging:**

- Overloading new can help you **track memory usage** (e.g., logging every allocation and deallocation) to help identify memory leaks or incorrect memory usage.

Performance Optimization

Tracking Memory Leaks:

- You can track every memory allocation and deallocation to detect memory leaks during the execution of your program.

Q7) difference between encapsulation and abstraction?

Ans) Encapsulation->Binding the data member and member function

- Here, Indirectly we are using Abstraction

Abstraction:-> hiding the implementation of our code (only essential we want to give code to client)

Q8) what is diamond problem?

Ans) **Diamond inheritance** is a term used in object-oriented programming to describe a scenario where a class inherits from two classes that have a common base class. This can create an ambiguous situation because the derived class ends up with two copies of the common base class, leading to potential conflicts and redundancy.

Here:

- Class B and class C both inherit from class A.
- Class D inherits from both B and C.

If class A has a method, for example display(), the question arises: which version of the display() method should class D inherit? This creates ambiguity and is referred to as the **diamond problem**.

C++ allows multiple inheritance, so this situation can arise. To solve the ambiguity, C++ provides **virtual inheritance**, ensuring that only one copy of the common base class is shared among derived classes.

```
class A {

public:

   void display() { std::cout << "Class A"; }

};


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

class D : public B, public C {};


int main() {

   D obj;

   obj.display();  // No ambiguity due to virtual inheritance

   return 0;

}
```

Q9) can we have virtual constructor?

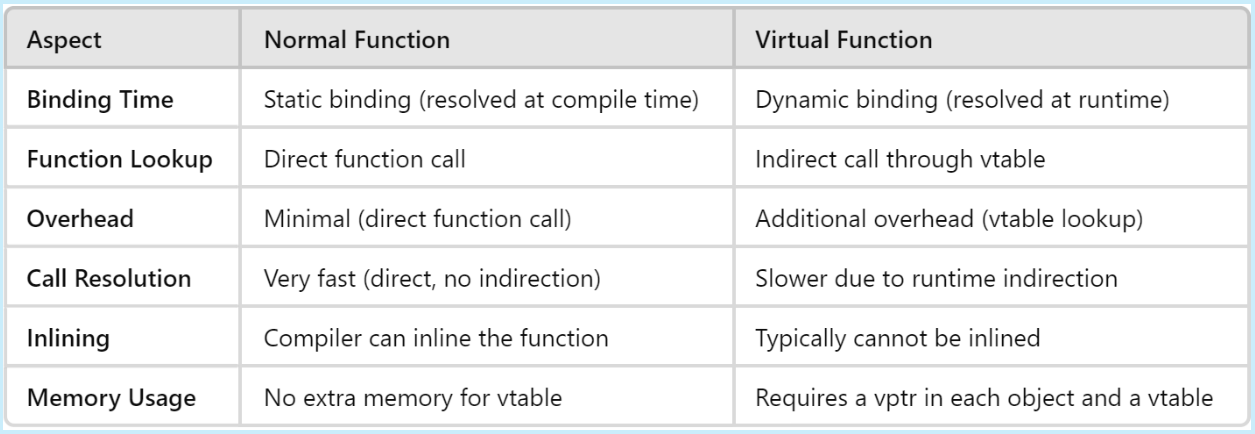
Ans) In C++, **we cannot have a virtual constructor**. This is because constructors are responsible for creating objects, and they are called during the object's initialization phase. Virtual functions, on the other hand, rely on the existence of a vtable (virtual table), which is only set up after the constructor completes. As a result, the concept of a "virtual constructor" is not supported in C++.

Q10) Can we have virtual destructor?

Ans) Yes, **C++ allows virtual destructors**, and they are essential for achieving proper polymorphic behavior when deleting objects through a base class pointer. A **virtual destructor** ensures that the destructors of derived classes are called correctly when an object is deleted, avoiding resource leaks or undefined behavior.

Q11) what is the difference between the performance of the normal function and the virtual function?

Ans->



Aspect	Normal Function	Virtual Function
Binding Time	Static binding (resolved at compile time)	Dynamic binding (resolved at runtime)
Function Lookup	Direct function call	Indirect call through vtable
Overhead	Minimal (direct function call)	Additional overhead (vtable lookup)
Call Resolution	Very fast (direct, no indirection)	Slower due to runtime indirection
Inlining	Compiler can inline the function	Typically cannot be inlined
Memory Usage	No extra memory for vtable	Requires a vptr in each object and a vtable

Q12) when will the vtable be created with respect to constructor ?

Ans-> The **vtable** will be created during the **Constructor** and the **vptr** will be assigned after the **completion of the Constructor** .

Q13) delete[] vs delete?

Q14)

Q15) try deep copy for a resource other than memory

Q16) Can I throw an exception from a destructor?

Ans-> Yes , we can throw but std:: terminate will be called and terminate the program . Good practice is to use noexcept.

Q17) . Can I throw an exception from a Constructor?

Ans-> Yes , we can throw exception from Constructor but then the object is partially created .

Q18) . What will happen if we dynamically allocate any memory in constructor before throw?

Ans) If memory is allocated in a constructor and an exception is thrown **before freeing that memory**, it may result in a **memory leak** unless proper cleanup is performed.

Memory Allocation in Constructor:

- When you allocate memory in a constructor (e.g., using new), it belongs to the object being constructed.
- If an exception is thrown before the constructor finishes, the object is not fully constructed, and the destructor will **not be called** for that object.

Q19). When do we use noexcept?

Ans) The noexcept specifier in C++ is used to indicate whether a function is guaranteed **not to throw exceptions**. It allows the compiler to perform optimizations and improves code reliability.

We have to use noexcept here:-

- Move Constructor/Move Assignment Operator
- getters and setters
- Destructor

Q20) . What is Stack Unwinding?

Q21) . A class is having another class object and then after allocating the memory to that object, it will throw error?

Q22) . Base ptr* = new Derived();

And wants to call the normal function from the derived class (throw dynamic cast).

Q23). What else can we do with the dynamic cast?? Downcast, upcast

Q24). How to achieve RTTI, how to identify the relationship between the class, Is A or Has A, can be done from dynamic cast.

Q25). What are the different ways of compile time polymorphism? (Template generic programming)

Q26). Runtime polymorphism ?

Q27). What could be the datatype of v table which will hold function pointers?

(Array of function pointers)

Q28). Why the empty class will take 1 byte?

Q29). How many vptr are there for a class, and how many vtable are there?

Q30). Function overriding is a compile time polymorphism, if there is a virtual then it is run time polymorphism.

Q31). Can we achieve Runtime polymorphism in C Programming?

Ans-> yes, through function pointers.

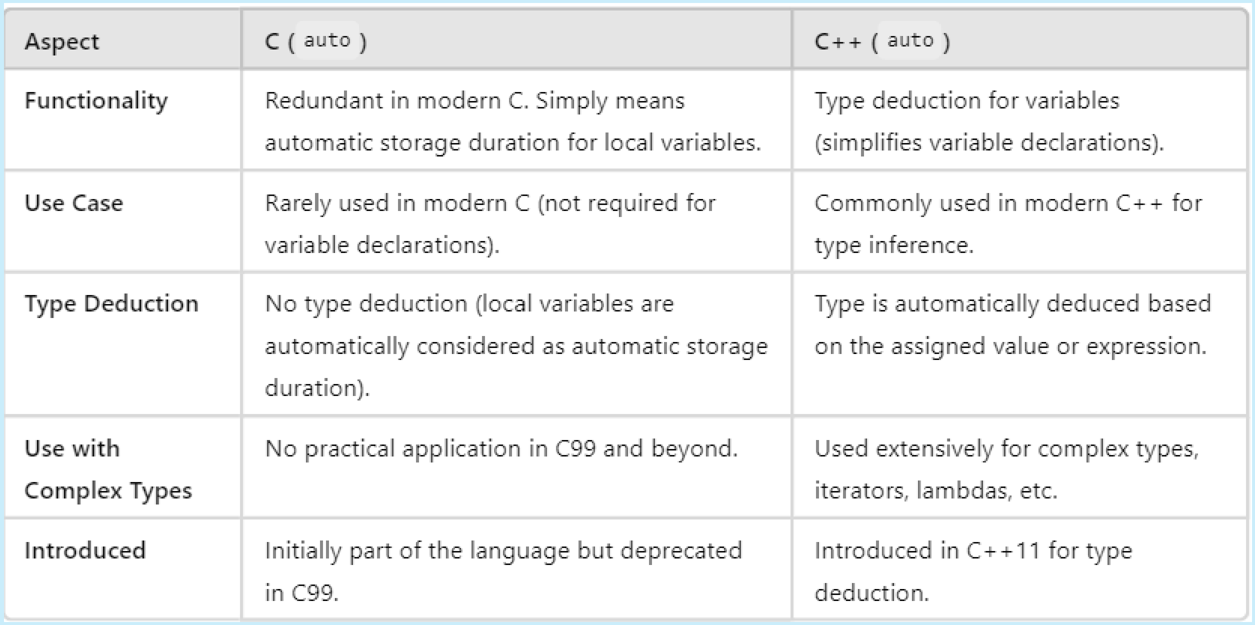
Q32). What are the disadvantages of Templates?

- Ans-> **Code Bloat**: Templates generate separate code for each instantiation, increasing binary size.
- **Compilation Overhead**: Templates lead to longer compile times due to code generation for each type.
- **Error Messages and Debugging Challenges**: Template-related errors are often cryptic and hard to debug.
- **Code Readability and Maintainability**: Templates can obscure logic and make code harder to read and maintain.
- **Template Specialization and SFINAE Complications**: Overusing or misusing specialization and SFINAE can increase code complexity.
- **Lack of Support for Non-Type Parameters**: Non-type template parameters are less intuitive and harder to use effectively.
- **Compatibility with Older Code**: Templates limit interoperability with older codebases or non-template-based libraries.
- **Potential for Logical Errors**: Template logic errors may only show up at compile time or with specific instantiations.
- **Reduced Compiler Optimization Opportunities**: Templates may prevent some compiler optimizations due to code generation for each instantiation.

Q33) Give Examples of different data structures real life implementations in different applications?

Q34) What is the difference between auto in c and cpp ?

Ans->

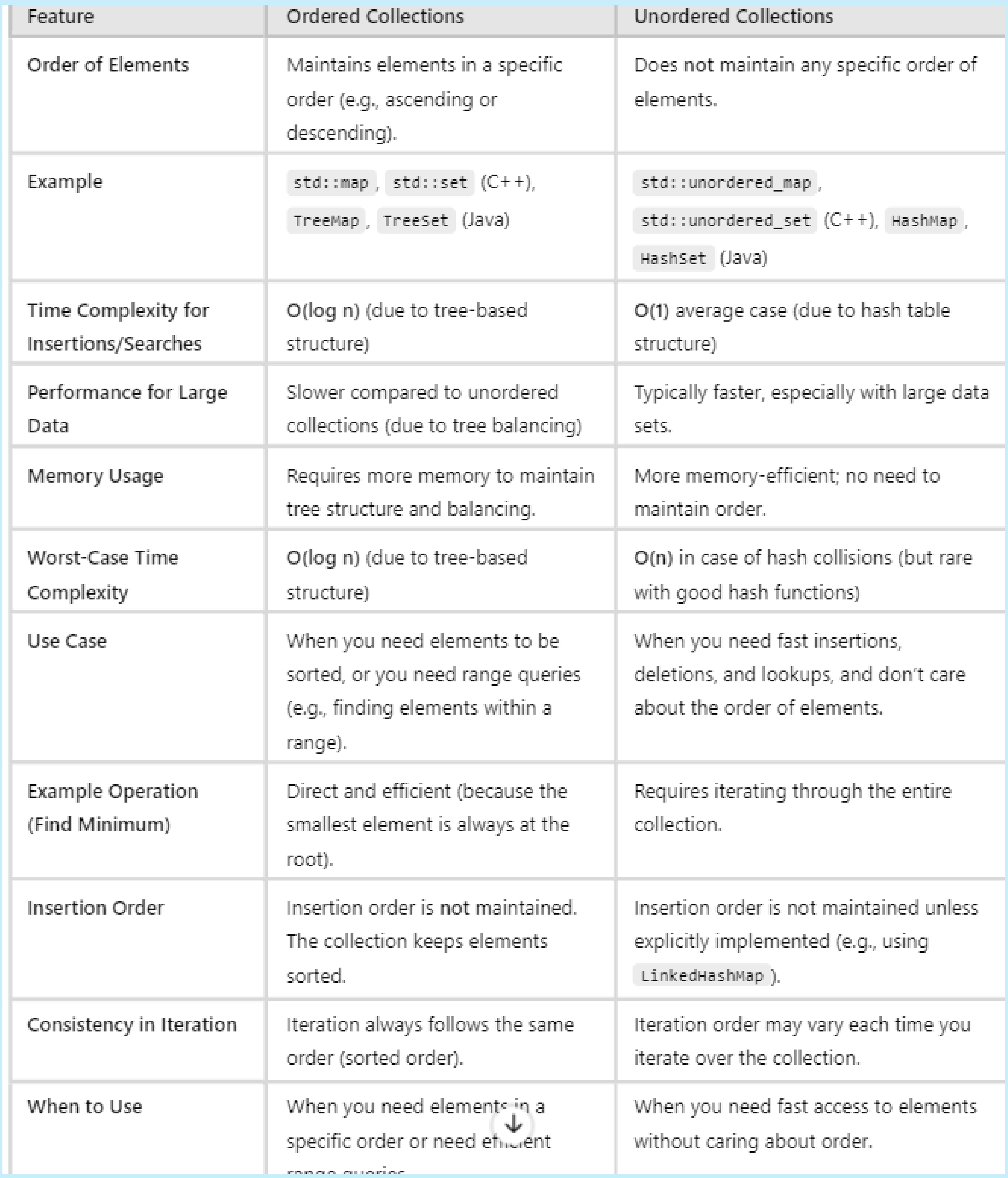


Aspect	C (`auto`)	C++ (`auto`)
Functionality	Redundant in modern C. Simply means automatic storage duration for local variables.	Type deduction for variables (simplifies variable declarations).
Use Case	Rarely used in modern C (not required for variable declarations).	Commonly used in modern C++ for type inference.
Type Deduction	No type deduction (local variables are automatically considered as automatic storage duration).	Type is automatically deduced based on the assigned value or expression.
Use with Complex Types	No practical application in C99 and beyond.	Used extensively for complex types, iterators, lambdas, etc.
Introduced	Initially part of the language but deprecated in C99.	Introduced in C++11 for type deduction.

Q35) What are Stable and Unstable algorithms ?

Q36) What is the difference between Ordered and Unordered containers?

Ans->



Feature	Ordered Collections	Unordered Collections
Order of Elements	Maintains elements in a specific order (e.g., ascending or descending).	Does not maintain any specific order of elements.
Example	`std::map`, `std::set` (C++), `TreeMap`, `TreeSet` (Java)	`std::unordered_map`, `std::unordered_set` (C++), `HashMap`, `HashSet` (Java)
Time Complexity for Insertions/Searches	O(log n) (due to tree-based structure)	O(1) average case (due to hash table structure)
Performance for Large Data	Slower compared to unordered collections (due to tree balancing)	Typically faster, especially with large data sets.
Memory Usage	Requires more memory to maintain tree structure and balancing.	More memory-efficient; no need to maintain order.
Worst-Case Time Complexity	O(log n) (due to tree-based structure)	O(n) in case of hash collisions (but rare with good hash functions)
Use Case	When you need elements to be sorted, or you need range queries (e.g., finding elements within a range).	When you need fast insertions, deletions, and lookups, and don't care about the order of elements.
Example Operation (Find Minimum)	Direct and efficient (because the smallest element is always at the root).	Requires iterating through the entire collection.
Insertion Order	Insertion order is not maintained. The collection keeps elements sorted.	Insertion order is not maintained unless explicitly implemented (e.g., using `LinkedHashMap`).
Consistency in Iteration	Iteration always follows the same order (sorted order).	Iteration order may vary each time you iterate over the collection.
When to Use	When you need elements in a specific order or need efficient range queries	When you need fast access to elements without caring about order.

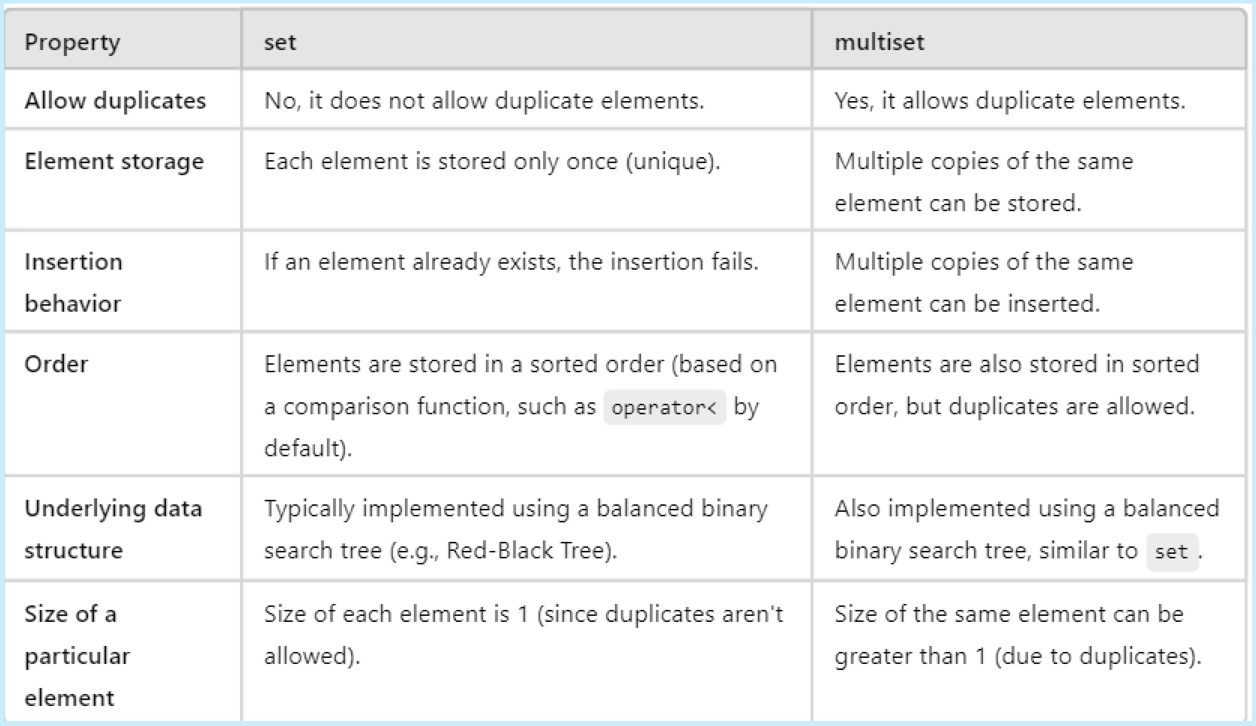
Q37) What is the thing we need to implement set class in cpp ?

Ans-> For unordered set – implement the less than operator AND

For ordered set – implement the equal to operator

Q38) What is the difference between set and multiset ?

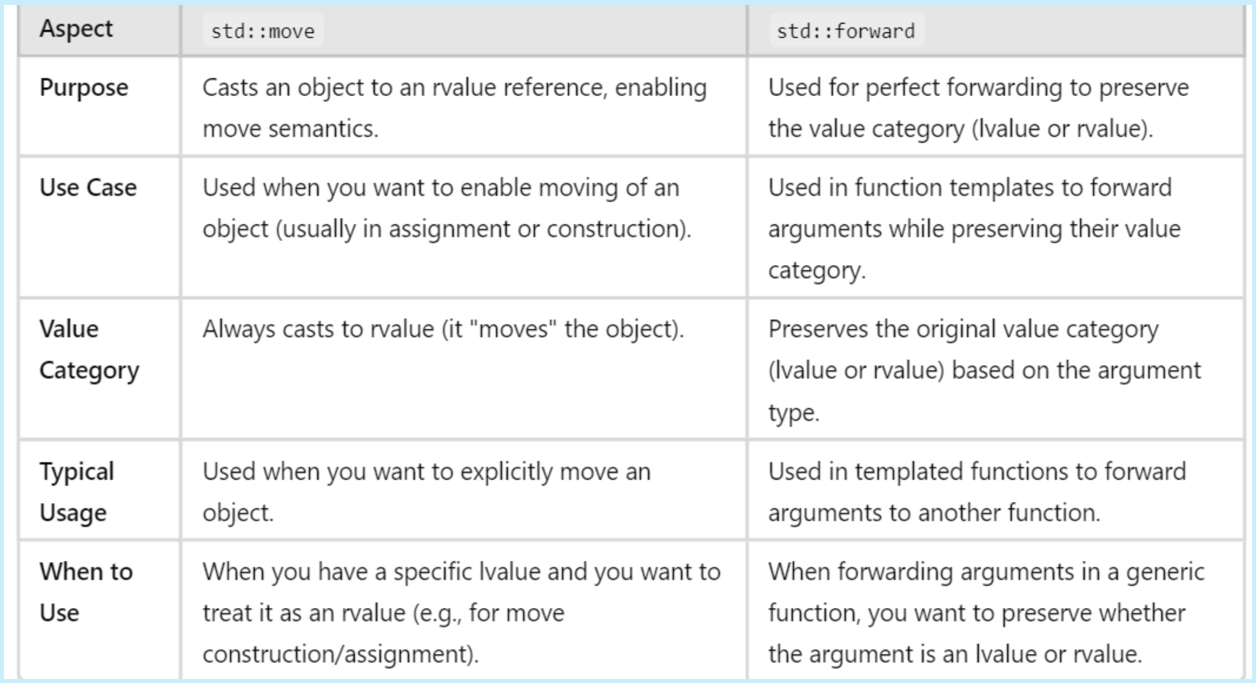
Ans->



Property	set	multiset
Allow duplicates	No, it does not allow duplicate elements.	Yes, it allows duplicate elements.
Element storage	Each element is stored only once (unique).	Multiple copies of the same element can be stored.
Insertion behavior	If an element already exists, the insertion fails.	Multiple copies of the same element can be inserted.
Order	Elements are stored in a sorted order (based on a comparison function, such as `operator<` by default).	Elements are also stored in sorted order, but duplicates are allowed.
Underlying data structure	Typically implemented using a balanced binary search tree (e.g., Red-Black Tree).	Also implemented using a balanced binary search tree, similar to `set`.
Size of a particular element	Size of each element is 1 (since duplicates aren't allowed).	Size of the same element can be greater than 1 (due to duplicates).

Q39) What is the difference between std::move and std::forward ?

Ans->



Aspect	`std::move`	`std::forward`
Purpose	Casts an object to an rvalue reference, enabling move semantics.	Used for perfect forwarding to preserve the value category (lvalue or rvalue).
Use Case	Used when you want to enable moving of an object (usually in assignment or construction).	Used in function templates to forward arguments while preserving their value category.
Value Category	Always casts to rvalue (it "moves" the object).	Preserves the original value category (lvalue or rvalue) based on the argument type.
Typical Usage	Used when you want to explicitly move an object.	Used in templated functions to forward arguments to another function.
When to Use	When you have a specific lvalue and you want to treat it as an rvalue (e.g., for move construction/assignment).	When forwarding arguments in a generic function, you want to preserve whether the argument is an lvalue or rvalue.

Q40)