Jadhav, Ajay - Dell Team

[Company name]  [Company address]

My OOPs Notes

Contents

[Design abstract classes or interface classes 2](#_Toc139450746)

[Use abstract classes when: 3](#_Toc139450747)

[Use interfaces when: 3](#_Toc139450748)

[Conclusion: 3](#_Toc139450749)

[Relationships: 4](#_Toc139450750)

[UML diagrams: 5](#_Toc139450751)

[Code Examples : 5](#_Toc139450752)

[significance of static\_cast vs dynamic\_cast in c++: 6](#_Toc139450753)

[**Type deduction and const references** 7](#_Toc139450754)

[**Member initializer lists:** 8](#_Toc139450755)

[**Delegating constructors** 8](#_Toc139450756)

[Virtual functions Rule: 8](#_Toc139450757)

[Covariant return types: 9](#_Toc139450758)

[Vtable 10](#_Toc139450759)

[Smart Pointers in C++ 10](#_Toc139450760)

[What is the C++ unique\_ptr? 10](#_Toc139450761)

[The shared\_ptr 11](#_Toc139450762)

[Weak\_ptr 11](#_Toc139450763)

[Guidelines to Follow When Working with Smart Pointers in C++ 12](#_Toc139450764)

[functor in c++ 12](#_Toc139450765)

# Design abstract classes or interface classes

The decision to design abstract classes or interface classes depends on the specific requirements and design goals of your application. Here are some considerations to help you make the choice:

## Use abstract classes when:

* You want to provide a common base implementation for a group of related classes.
* You want to define default behavior for certain methods.
* You need to access protected members or fields within the hierarchy of related classes.
* You want to create a class hierarchy that represents a "is-a" relationship, where subclasses are more specific types of the abstract class.
* You anticipate the need to add new methods or members in the future, while still providing a default implementation for existing methods.

## Use interfaces when:

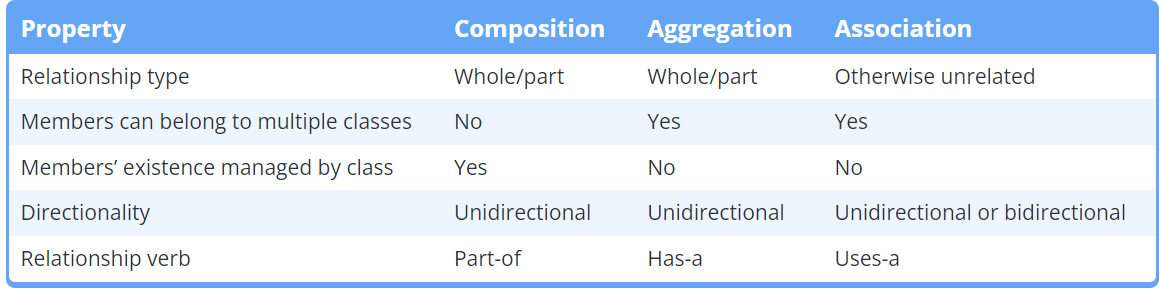
* You want to define a contract or a set of method signatures that classes should implement, regardless of their inheritance hierarchy.
* You need to support multiple inheritance-like behavior, where a class can implement multiple interfaces.
* You want to **enable loose coupling between classes**, allowing different implementations to be easily swapped.
* You have unrelated classes that need to share common behavior.
* You want to enforce a certain level of abstraction and ensure adherence to a specific interface.

### Conclusion:

It's important to note that abstract classes and interfaces can be used together in a design. For example, you might have an abstract class providing a base implementation and implementing an interface that defines additional behavior. This combination can provide flexibility while maintaining a clear contract through interfaces.

Ultimately, the decision between abstract classes and interfaces depends on the specific needs and goals of your application's architecture, the relationship between classes, and the level of flexibility and abstraction required.

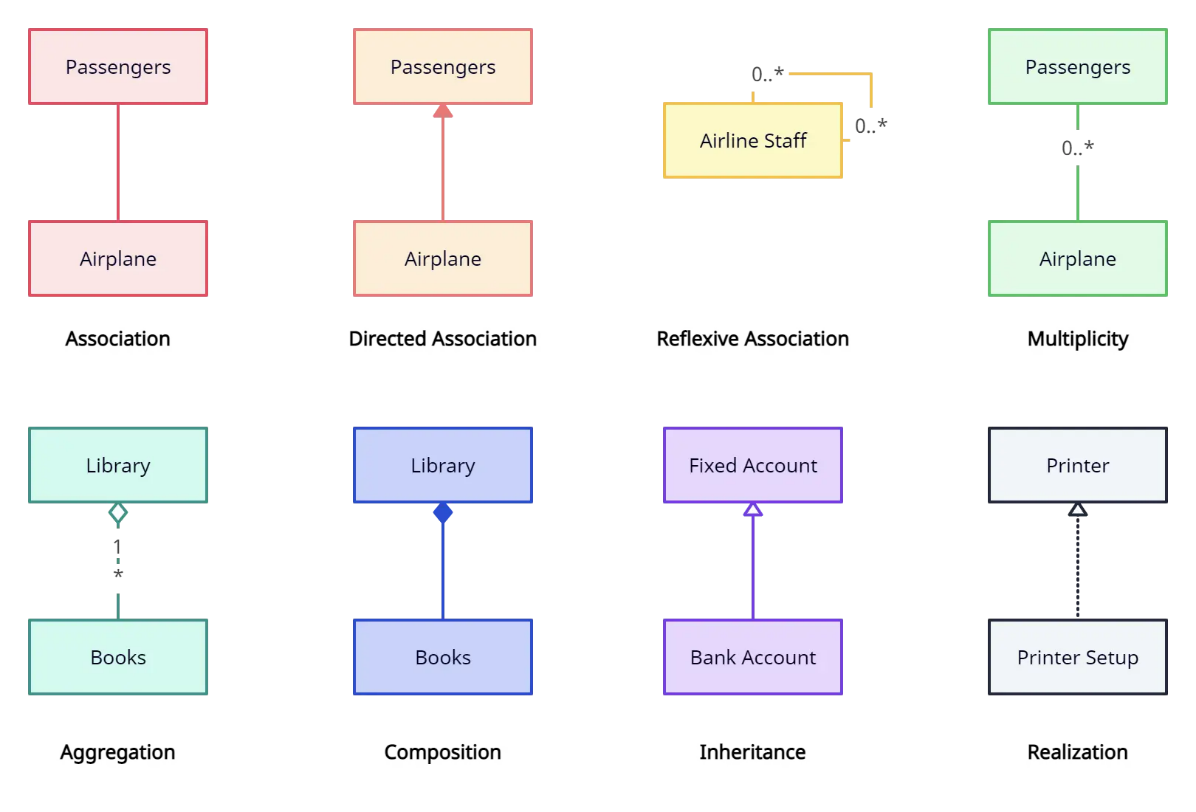
# Relationships:



Remember association with verb relationship among two classes like Dog eats, Driver driving car

a reflexive association relationship between the University class and the Student class. This means that each Student object is associated with a University object, and each University object maintains a collection of Student objects.

## UML diagrams:



## Code Examples :

class Car {

private:

string make;

Engine engine; // Composition relationship

Driver\* driver; // Association relationship

public:

Car(const string& carMake, const string& engineType, Driver\* carDriver)

: make(carMake), engine(engineType), driver(carDriver) {}

void startCar() {

cout << "Starting the " << make << " car." << endl;

engine.start();

driver->driveCar(make);

}

// Aggregate class

class Car {

private:

string make;

Engine\* engine; // Aggregation relationship

public:

Car(const string& carMake, Engine\* carEngine) : make(carMake), engine(carEngine) {}

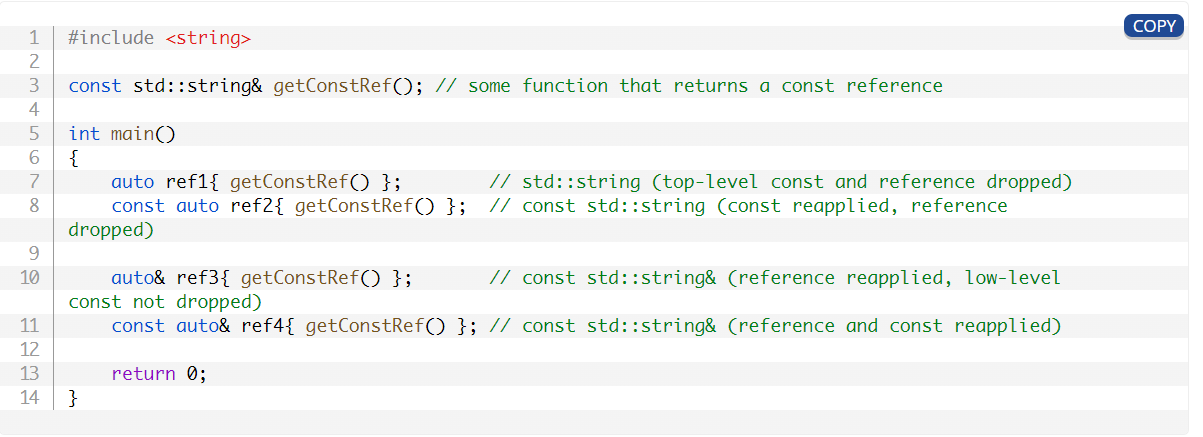
In summary, the association between the **University** and **Department** represents **a logical relationship, indicating their collaboration.** The aggregation relationship denotes that the **University contains multiple Department objects as part of its structure,** but the lifecycles of the two classes are not tightly coupled.

# significance of static\_cast vs dynamic\_cast in c++:

**static\_cast** and **dynamic\_cast** in C++:

|  |  |  |
| --- | --- | --- |
|  | static\_cast | dynamic\_cast |
| Purpose | Safe and implicit conversions, explicit type conversions | Safe conversions involving polymorphic types |
| Compile-time | Yes | No |
| Runtime type checks | No | Yes |
| Efficiency | More efficient (no runtime checks) | Slightly slower (due to runtime type checks) |
| Implicit conversions | Yes | No |
| Safety guarantees | Lacks runtime safety guarantees | Provides runtime type checking for safe conversions |
| Casting way | Upcasting Allowed, assuming conversion is valid | Allowed, ensures type safety during **downcasting** |
| Null pointer or exception | No (may lead to undefined behavior) | Null pointer (for pointer conversions), **std::bad\_cast** exception (for reference conversions) |
| Usage | Known and predictable conversions | Runtime type verification, conversions within inheritance hierarchy |
| Examples | Numeric conversions, pointer conversions within inheritance hierarchy | Polymorphic type conversions, downcasting |

# **Type deduction and const references**



# **Member initializer lists:**

Member initializer lists allow us to initialize our members rather than assign values to them. This is the only way to initialize members that **require values upon initialization, such as const or reference members, and it can be more performant than assigning values in the body of the constructor.** Member initializer lists work both with fundamental types and members that are classes themselves.

# **Delegating constructors**

Constructors are allowed to call other constructors from the same class. This process is called **delegating constructors** (or **constructor chaining**).

# Virtual functions Rule:

* If a function is virtual, **all matching overrides** in derived classes are implicitly virtual.
* This does not work the other way around -- a virtual override in a derived class does not implicitly make the base class function virtual.
* **Best practice**
* Never call virtual functions from constructors or destructors.

**BaseClass**\* obj = new DerivedClass();

// Here, the static type of 'obj' is **BaseClass**, even though it points to an instance of DerivedClass.

BaseClass\* obj = new **DerivedClass**();

// Here, the dynamic type of the object being pointed to is **DerivedClass**.

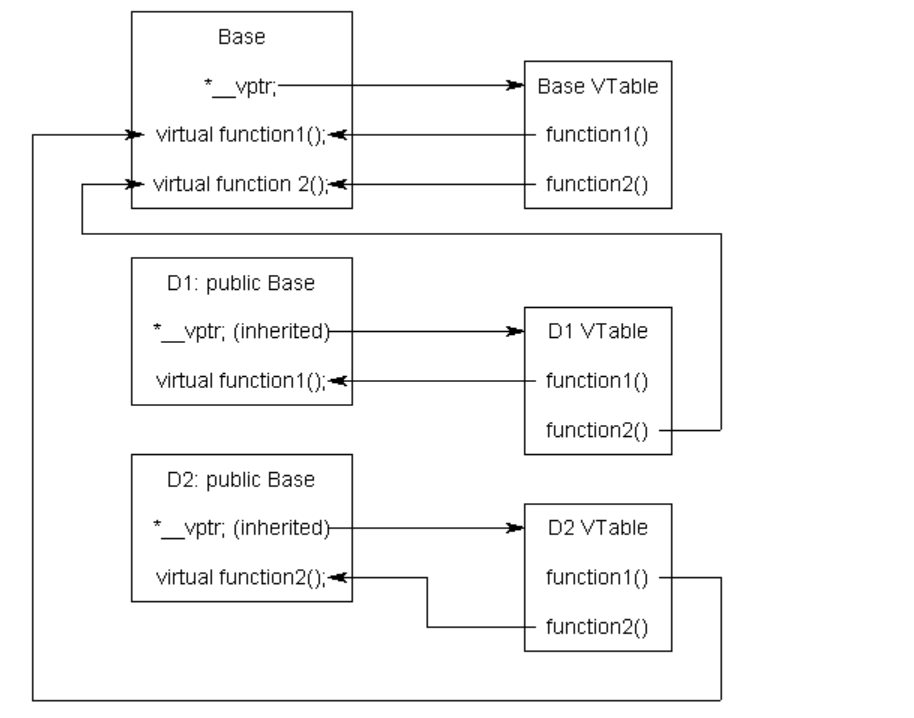
// If 'obj' calls an overridden method, the method in DerivedClass will be invoked.

# Covariant return types:



Note that if printType() were virtual instead of non-virtual, the result of b->getThis() (an object of type Base\*) would have undergone virtual function resolution, and Derived::printType() would have been called.

Vtable:



# Smart Pointers in C++

smart\_pointer\_type<data\_type>pointer\_name(new data\_type())*;*

## What is the C++ unique\_ptr?

* std::unique\_ptr was introduced in C++11. Unique\_ptr is a container that holds raw pointers and **prevents copying of that pointer.**
* There can only be one unique\_ptr per resource. It is not possible to copy one unique\_ptr to another since one unique\_ptr can only belong to a single resource.
* It is possible to move one unique\_ptr to another unique\_ptr by using std::move().
* If the allocated object on the heap is destroyed if the unique pointer is destroyed.
* The unique\_ptr can be misused if in the program more than one class manages the same resource or the resource is manually deleted from underneath the std::unique\_ptr.
* unique\_ptr<class\_name> object\_name (new class\_name);

{

unique\_ptr<int> p(new int);

*// using p*

} *// destructing p destroys the int object.*

The shared\_ptr allows you to make a copy of the pointer. It will hold the memory until all the pointer holding that memory gets out of scope. This is done by maintaining a **reference counter.**

The Reference counter will hold the count of pointers pointing to that memory location. The destructor will check the reference counter and free the memory only if **the reference counter value is 1**, i.e., only the current pointer is pointing to that memory.

You can also revoke the ownership you hold over the memory that the pointer holds with the reset method. In that case, the reference count will be decreased by 1, representing that there is one less owner for that memory location. At any point in time if you need to know the number of pointers pointing to a location you can use the **use\_count()** function to get that.

## Weak\_ptr

The weak\_ptr is much like the shared\_ptr. The only difference is that if you created a **weak\_ptr to a shared\_ptr, the reference count would not increase.**

So The Smart Pointer will **free the memory irrespective of whether the weak\_ptr is still in scope.** The Programmer can use this if you want to check whether the shared\_ptr still holds the memory or not.

# Guidelines to Follow When Working with Smart Pointers in C++

* Always try to use smart pointers in C++ because it is better to be sure that you **won't be running out of memory due to a memory leak.**
* Use Unique\_ptr if **you are not sure to use** which of the smart pointer to use.
* Use Shared\_ptr only when **dealing with multiple pointers or threads** where you will need to share the location.
* If you want just to examine an object and **not gonna work with it on any serious level, you can then use the Weak\_ptr.**
* Try to reduce the usage of raw pointers as much as possible, and when you do, make sure that you properly free the memory held by the pointer.

# functor in c++

In C++, a functor is **an object that can be treated as a function**. Functors are instances of classes or structs that **overload the function call operator operator()** and can be invoked like regular functions.

    int operator()(int x)

# push\_back vs emplace\_back

push\_back **copies or moves an existing object into the vector**, while emplace\_back **constructs the object in place within the vector itself**.

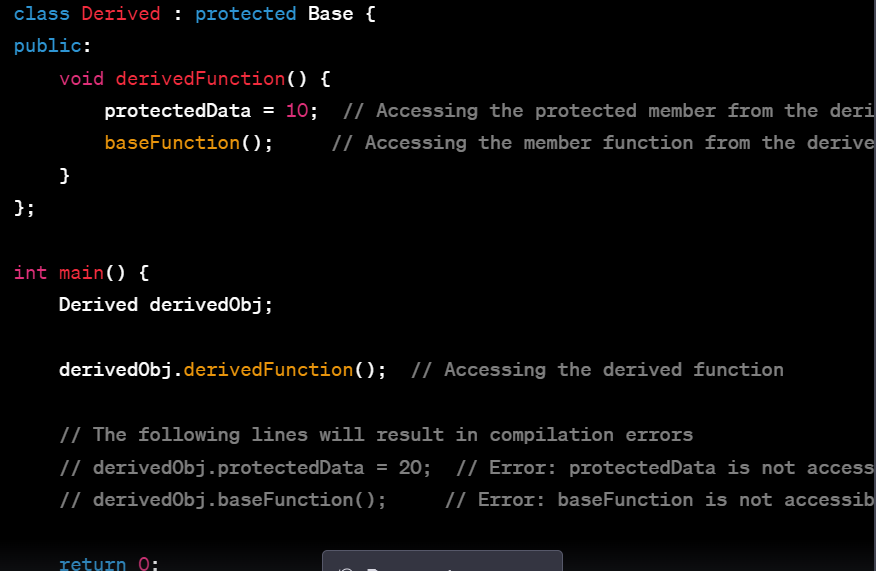
## When to use:

If you have a pre-existing object or want to provide explicit constructor arguments, push\_back is appropriate.

However, if you want to **construct the object directly within the vector using constructor arguments,** emplace\_back offers a more efficient alternative.

# Protected inheritance

Protected inheritance is a type of inheritance in C++ where the derived class inherits the members of the base class with the protected access specifier. This means that the derived class and its derived classes can access the inherited members, **but the members are not accessible outside the class hierarchy.**



# Most Used STL Algos:

**std::sort:** Sorts elements in a range into ascending order.

**std::binary\_search:** Checks if a value exists in a sorted range.

**std::find:** Finds the first occurrence of a value in a range.

**std::count:** Counts the occurrences of a value in a range.

**std::accumulate:** Computes the sum/diff/mod of a range of values.

**std::max\_element:** Finds the maximum element in a range.

**std::min\_element:** Finds the minimum element in a range.

**std::reverse:** Reverses the order of elements in a range.

**std::transform:** Applies a function to each element in a range and stores the result.

**std::copy:** Copies elements from a source range to a destination range.

**std::remove:** Removes elements satisfying a given value from a range.

**std::unique:** Removes consecutive duplicate elements from a range.

**std::partition:** Partitions a range into elements that satisfy a given condition.

**std::merge:** Merges two sorted ranges into a single sorted range.

**std::find\_if:** Finds the first element in a range that satisfies a given condition.

**std::replace:** Replaces all occurrences of a value in a range with another value.

**std::swap:** Swaps the values of two objects.

# std::find and vector::find

std::find and vector::find are two different functions that serve similar purposes but have different implementations and usage.

The **vector::find** function internally uses the **std::find** algorithm to perform the search.

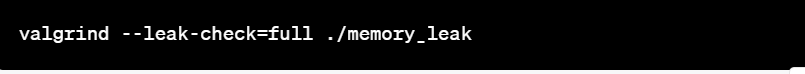
# the mutable keyword

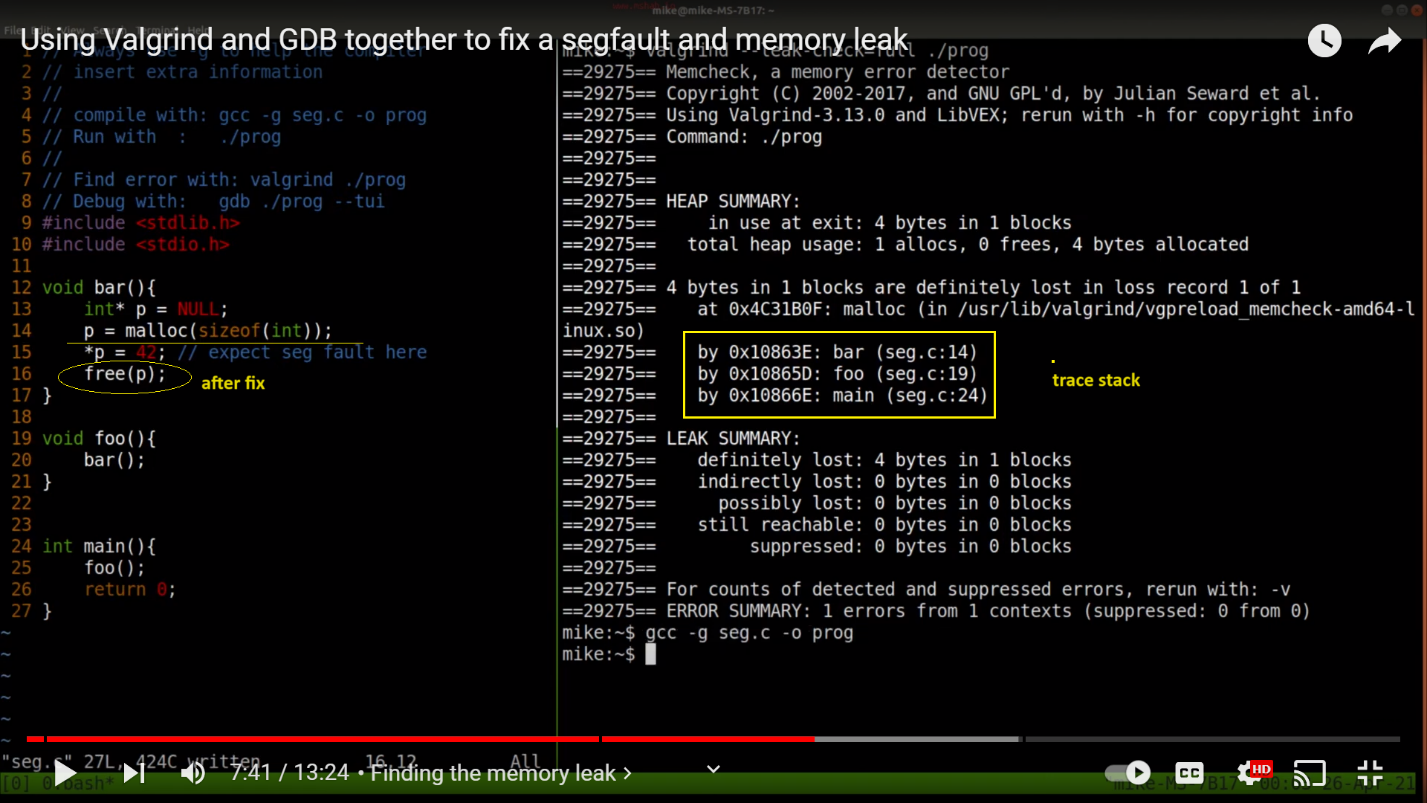
the mutable keyword is used to modify the behavior of a non-static data member of a class. When you declare a data member as mutable, it means that **even if an object of the class is declared as const, you can still modify the mutable member within a const member function of that class.** This allows you to change the internal state of an object even when it is considered "constant" from the outside.

The primary use case for mutable is when you have a data member that logically represents some internal cached or temporary state within an object but does not alter its external observable behavior.

# Detect memory leak using Valgrind

valgrind --leak-check=full ./memory\_leak





The line **All heap blocks were freed -- no leaks are possible** indicates that Valgrind did not find any memory leaks in your program.

# **malloc/calloc and new/delete:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **malloc/calloc (C)** | **new/delete (C++)** |
| **Purpose** | Allocate raw memory | Allocate memory for objects |
| **Memory Initialization** | Uninitialized | Calls constructors (initialized) |
| **Memory Deallocation** | **free** | **delete** |
| **Handling Objects** | Does not call constructors or destructors | Calls constructors and destructors |
| **Array Allocation** | **malloc** for memory blocks, **calloc** for zero-initialized memory blocks | **new** for single objects, **new[]** for arrays |
| **Array Deallocation** | **free** for memory blocks | **delete** for single objects, **delete []** for arrays |
| **Type Safety** | Not type-safe, requires manual casting | Type-safe, aware of object types |
| **Compatibility with C** | Compatible with C code | C++ specific, not compatible with C |
| **Error Handling** | Returns **NULL** on failure | Throws **std::bad\_alloc** on failure |
| **Usage** | int \*arr = (int \*)malloc(5 \* sizeof(int));  int \*arr = (int \*)calloc(5, sizeof(int)); | int \*arr = new int[5];  // ...  delete[] arr; |

# How delete[] Knows How Much To Deallocate In C++?

Delete [] ptrs;

[Size Information][Element 1][Element 2]...[Element N]

* **Size Information** is stored internally by the C++ runtime and represents the size of the array.
* **Element 1**, **Element 2**, ..., **Element N** are the actual elements of the array.

When you call **delete []**, it knows to deallocate the entire block of memory, including **Size Information** and all elements, based on the size information stored internally.

# Alternative Operators In C++

C++ operator synonyms[[edit](https://en.wikipedia.org/w/index.php?title=Operators_in_C_and_C%2B%2B&action=edit&section=12)]

C++ defines[[17]](https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#cite_note-Committee-33) certain keywords to act as aliases for a number of operators:

|  |  |
| --- | --- |
| **Keyword** | **Operator** |
| and | && |
| and\_eq | &= |
| bitand | & |
| bitor | | |
| compl | ~ |
| not | ! |
| not\_eq | != |
| or | || |
| or\_eq | |= |
| xor | ^ |
| xor\_eq | ^= |

# Explicit constructor In C++

class MyClass {

public:

    MyClass(int value) {

        // Constructor code

    }

};

MyClass obj = 42; // Error: Cannot convert int to MyClass explicitly

MyClass obj(42); // Explicit constructor call

# Placement new In C++

    // Create a memory buffer to hold a MyClass object

    char buffer[sizeof(MyClass)];

    // Use placement new to create a MyClass object in the buffer

    MyClass\* obj = new (buffer) MyClass(42);

// Explicit destructor call we must need an object here to call destru

obj->~MyClass();

Placement **new** is a powerful tool for precise memory management and is often used in advanced scenarios where **objects need to be placed in specific memory locations**. However, it should be used with care to ensure that memory is properly allocated and deallocated, and destructors are explicitly called when necessary.

We can call **the destructor explicitly for objects created using placement new** to ensure that resources are properly released. This is necessary because placement new does not automatically call destructors.

# What Is Constructor Delegation In C++?

Constructor delegation in C++ refers to the ability of one constructor within a class to call another constructor of the same class. This allows you to reuse constructor code and initialize objects in different ways without duplicating common initialization logic.

    // Primary constructor

    MyClass(int value) : value\_(value)

    {

        std::cout << "Primary constructor called with value: " << value << std::endl;

    }

    // Delegating constructor that calls the primary constructor

    MyClass() : MyClass(0)

    {

        std::cout << "Delegating constructor called with default value" << std::endl;

    }

# extern "C" In C++

By using **extern "C",** you make C++ code compatible with C linkage, making it easier to interface with C libraries and ensuring that C++ code can be called from C code without issues related to name mangling and calling conventions.

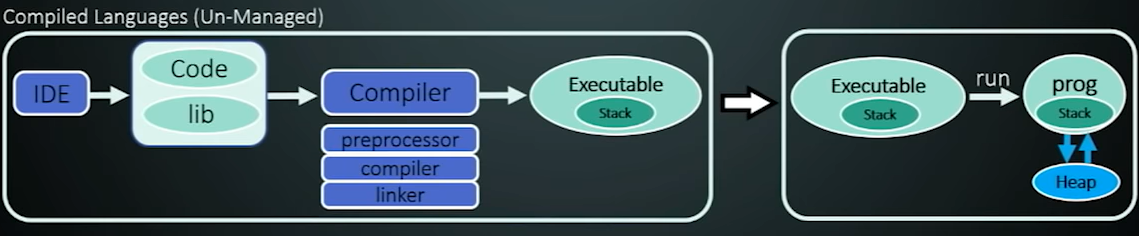
# Is It Possible To Call constructor And destructor Explicitly?

Yes, b

# Move Semantics:

* Moving an object means transferring its resources to another object while leaving the source object in a valid but unspecified state. Make sure you're aware of **what "moving" actually means in terms of object state.**
* **Immutable Objects**: Objects marked as **const** or objects whose internal state should remain unchanged should not be moved. Attempting to move a **const** object may compile, but it won't change the object's state and can lead to unexpected behavior.
* **Move-Only Types**: Some objects may be designed to be **move-only, meaning they cannot be copied.** In such cases, provide a move constructor while disabling the copy constructor to prevent unintentional copies. For exe, **unique\_ptr is move-only type.**
* **Use std::move\_if\_noexcept**: When using move semantics, consider using **std::move\_if\_noexcept** to optimize move operations in a way that avoids potential exceptions. This can improve performance when dealing with types that have a strong exception guarantee.

**Ownership**

****

# Const Object and Const function

1. **Const Object**:

A const object is an instance of a class or a variable that is marked as **const**. When an object is declared as **const**, it means that its state cannot be modified after its initialization. The key characteristics of const objects are:

* + You can only call const member functions on const objects. This is because **const objects are considered read-only and calling a non-const member** function would imply the potential for modification of the object's state.
  + Any attempt to modify a member variable of a const object will result in a **compilation error.**

1. **Const Member Function**:

A const member function is a function within a class that is declared as **const**. This means that **the function is not allowed to modify the state of the object on which it is called**. The key characteristics of const member functions are:

* + They can only access member variables of the class that are also marked as **const** or are declared as **mutable**.
  + They can **be called on both const and non-const objects**.