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:

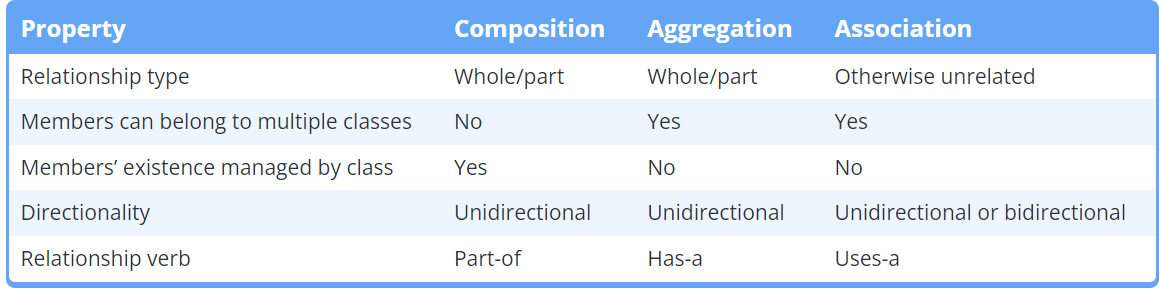
* You want to define a contract or a set of method signatures that classes should implement, regardless of their inheritance hierarchy.
* You need to support multiple inheritance-like behavior, where a class can implement multiple interfaces.
* You want to **enable loose coupling between classes**, allowing 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.

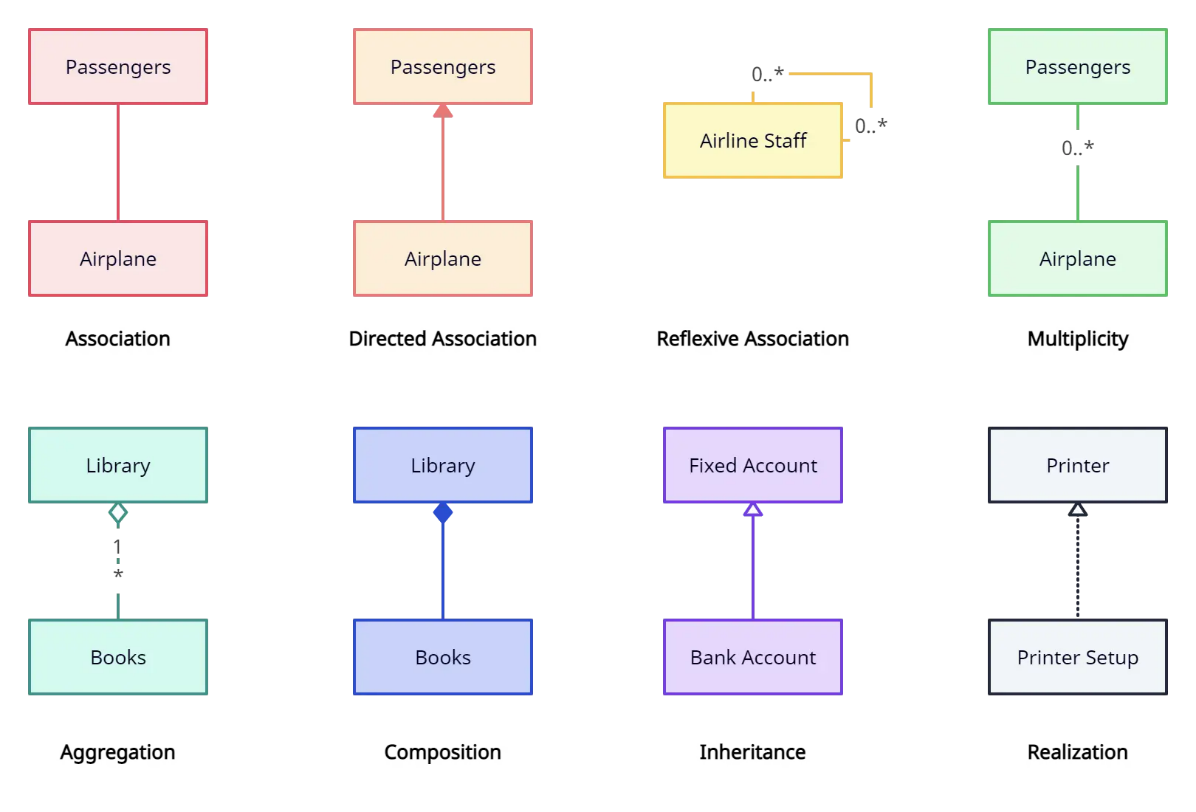
# Relationships:



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) {}

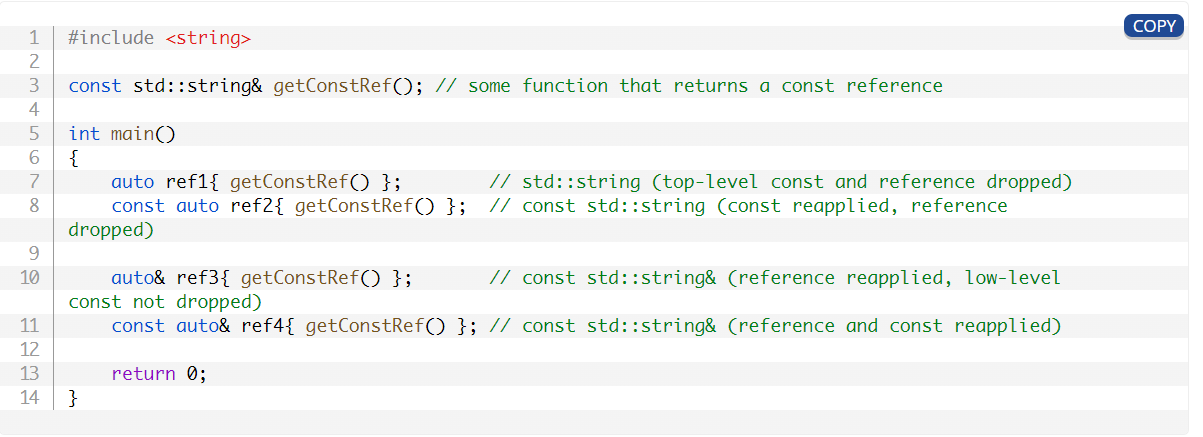
In summary, the association between the **University** and **Department** represents **a logical relationship, indicating their collaboration.** The aggregation relationship denotes that the **University contains multiple Department objects as part of its structure,** but the lifecycles of the two classes 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:

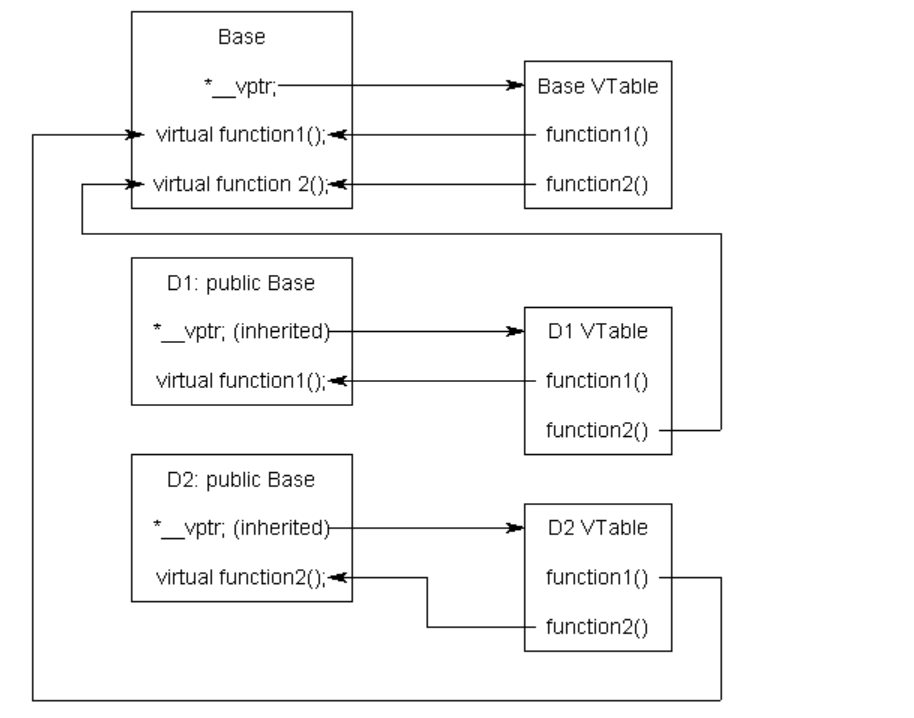
* If a function is virtual, **all matching overrides** in derived classes are implicitly virtual.
* This does not work the other way around -- a virtual override in a derived class does not implicitly make the base class function virtual.
* **Best practice**
* Never call virtual functions from constructors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

# 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.

**Summary**

The way that the access specifiers, inheritance types, and derived classes interact causes a lot of confusion. To try and clarify things as much as possible:

First, a class (and friends) can always access its own non-inherited members. The access specifiers only affect whether outsiders and derived classes can access those members.

Second, when derived classes inherit members, those members may change access specifiers in the derived class. This does not affect the derived classes’ own (non-inherited) members (which have their own access specifiers). It only affects whether outsiders and classes derived from the derived class can access those inherited members.

Here’s a table of all of the access specifier and inheritance types combinations:

|  |  |  |  |
| --- | --- | --- | --- |
| **Access specifier in base class** | **Access specifier when inherited publicly** | **Access specifier when inherited privately** | **Access specifier when inherited protectedly** |
| Public | Public | Private | Protected |
| Protected | Protected | Private | Protected |
| Private | Inaccessible | Inaccessible | Inaccessible |

As a final note, although in the examples above, we’ve only shown examples using member variables, these access rules hold true for all members (e.g. member functions and types declared inside the class).

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 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 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.