Jadhav, Ajay - Dell Team

[Company name]  [Company address]

My OOPs Notes

Contents

[Design abstract classes or interface classes 2](#_Toc138690510)

[Use abstract classes when: 2](#_Toc138690511)

[Use interfaces when: 2](#_Toc138690512)

[Conclusion: 2](#_Toc138690513)

[Relationships: 3](#_Toc138690514)

[UML diagrams: 4](#_Toc138690515)

[Code Examples : 4](#_Toc138690516)

[significance of static\_cast vs dynamic\_cast in c++: 5](#_Toc138690517)

[**Type deduction and const references** 6](#_Toc138690518)

[**Member initializer lists:** 6](#_Toc138690519)

[**Delegating constructors** 6](#_Toc138690520)

# 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:

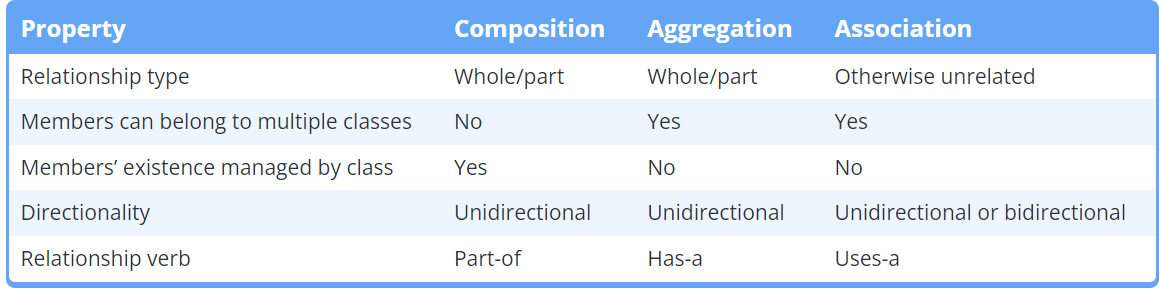
* You want to define a contract or a set of method signatures that classes should implement, regardless of their inheritance hierarchy.
* You need to support multiple inheritance-like behavior, where a class can implement multiple interfaces.
* You want to enable loose coupling between classes, allowing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

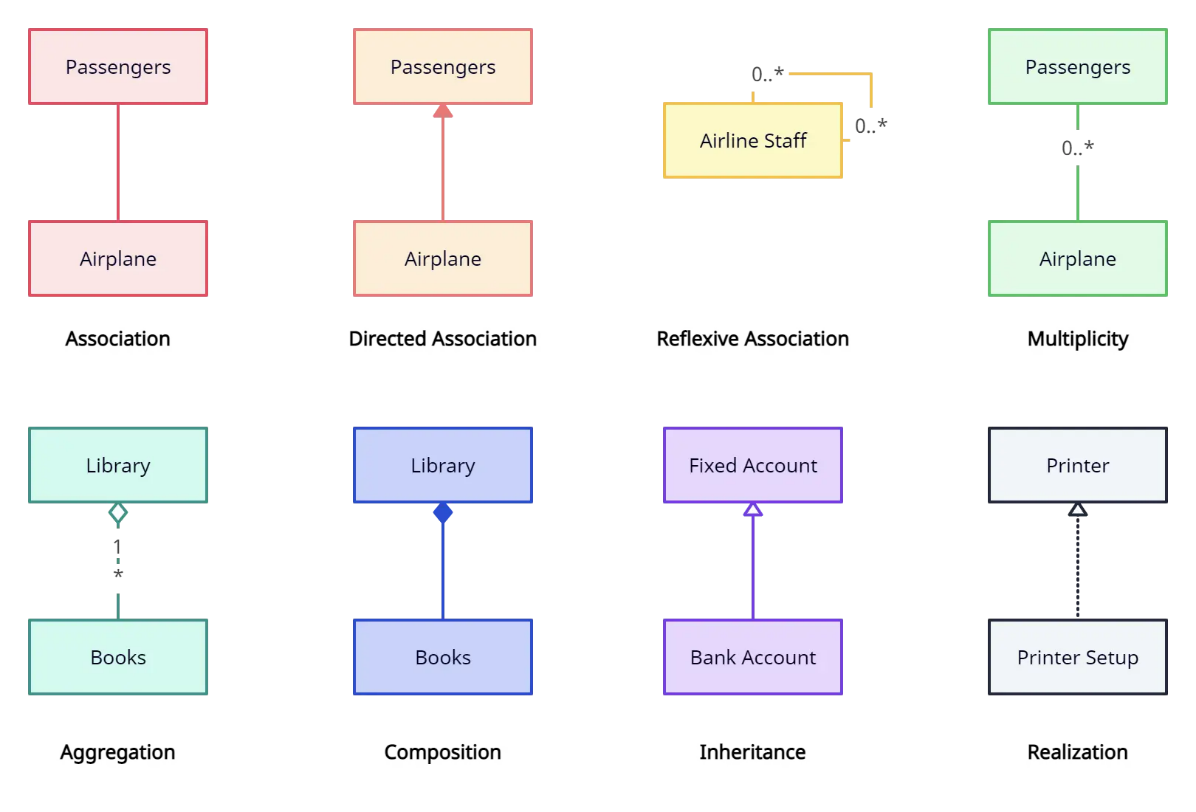
# Relationships:



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:

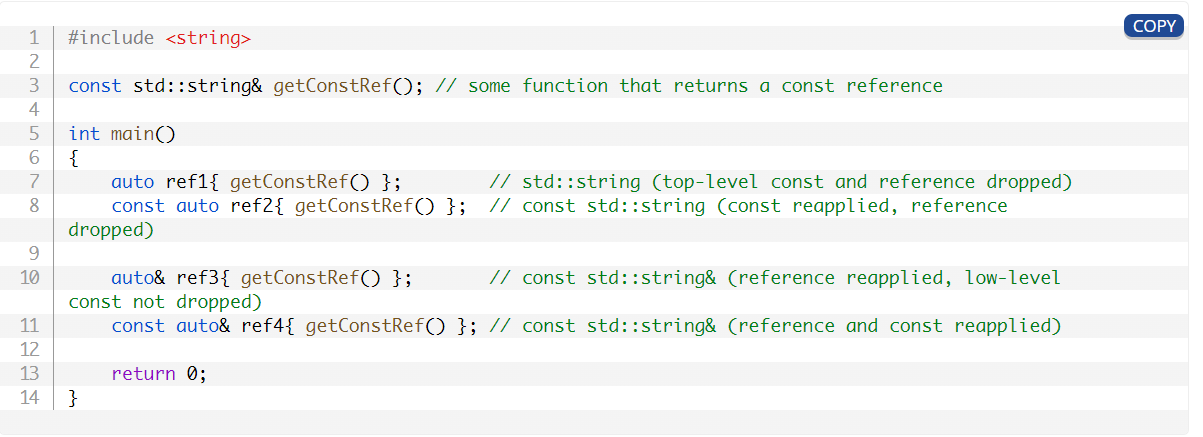
Car(const string& carMake, Engine\* carEngine) : make(carMake), engine(carEngine) {}

# 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 |
| Downcasting | 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**).