Inheritance

Dr. Anwar Shah, PhD, MBA(HR)

Assistant Professor in CS

FAST National University of Computer and Emerging Sciences CFD

Section Overview

Inheritance

- What is Inheritance?
  - Why is it useful?
- Terminology and Notation
- Inheritance vs. Composition
- Deriving classes from existing classes
  - Types of inheritance
- Protected members and class access
- Constructors and Destructors
  - Passing arguments to base class constructors
  - Order of constructor and destructors calls
- Redefining base class methods
- Class Hierarchies
- Multiple Inheritance

What is Inheritance?

Inheritance

What is it and why is it used?

- Provides a method for creating new classes from existing classes
- The new class contains the data and behaviors of the existing class
- Allow for reuse of existing classes . It is possible because of the relationship exists between the classes.
- Allows us to focus on the common attributes among a set of classes
- Allows new classes to modify behaviors of existing classes to make it unique
    - Without actually modifying the original class

Inheritance

Related classes

- Player, Enemy, Level Boss, Hero, Super Player, etc.
- Account, Savings Account, Checking Account, Trust Account, etc.
- Shape, Line, Oval, Circle, Square, etc.
- Person, Employee, Student, Faculty, Staff, Administrator, etc.

Inheritance

Accounts

- Account
  - **balance**, **deposit**, **withdraw**, . . .
- Savings Account
  - **balance**, **deposit**, **withdraw**, interest rate, . . .
- Checking Account
  - **balance**, **deposit**, **withdraw**, minimum balance, per check fee, . . .
- Trust Account
  - **balance**, **deposit**, **withdraw**, interest rate, . . .

Inheritance

Accounts – without inheritance – code duplication

```
class Account {
    // balance, deposit, withdraw, . . .
};


class Savings_Account {
    // balance, deposit, withdraw, interest rate, . . .
};

class Checking_Account {
    // balance, deposit, withdraw, minimum balance, per check fee, . . .
};

class Trust_Account {
    // balance, deposit, withdraw, interest rate, . . .
};
```

Inheritance

Accounts – with inheritance – code reuse

```
class Account {
    // balance, deposit, withdraw, . . .
};

class Savings_Account : public Account {
    // interest rate, specialized withdraw, . . .
};

class Checking_Account : public Account {
    // minimum balance, per check fee, specialized withdraw, . . .
};

class Trust_Account : public Account {
    // interest rate, specialized withdraw, . . .
};
```

Terminology and Notation

Inheritance

Terminology

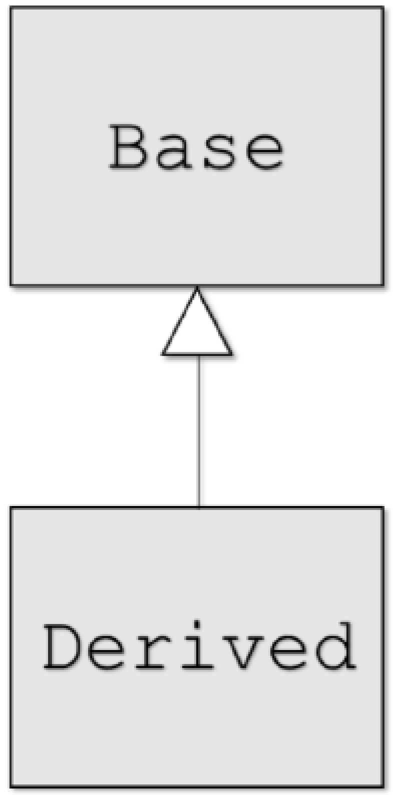
- Inheritance
  - Process of creating new classes from existing classes
  - Reuse mechanism
- Single Inheritance
  - A new class is created from another ‘single’ class
- Multiple Inheritance
  - A new class is created from two (or more) other classes

Inheritance

Terminology

- Base class (parent class, super class)
  - The class being extended or inherited from

- Derived class (child class, sub class)
  - The class being created from the Base class
  - Will inherit attributes and operations from Base class



Inheritance

Terminology

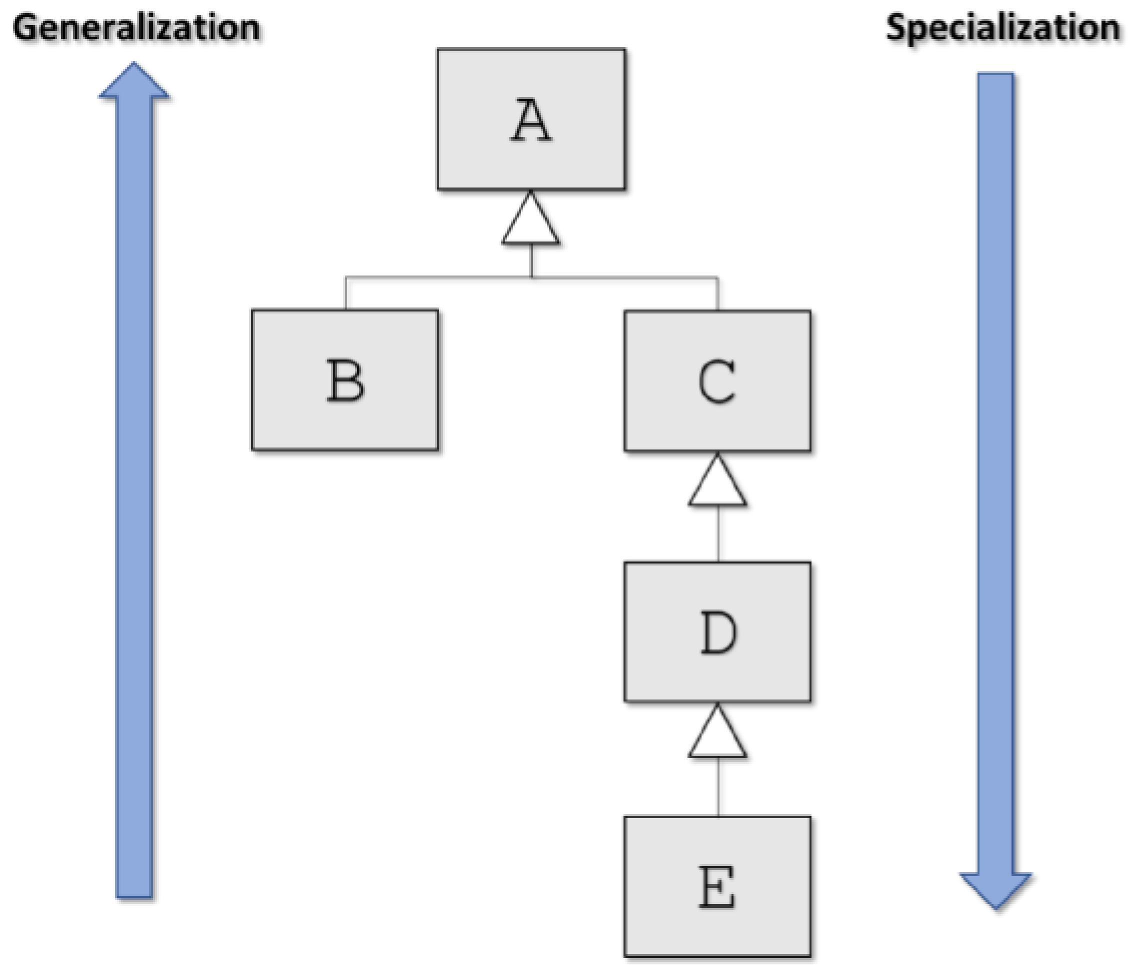
- "Is-A" relationship
  - Public inheritance
  - Derived classes are sub-types of their Base classes
  - Can use a derived class object wherever we use a base class object
- Generalization
  - Combining similar classes into a single, more general class based on common attributes
- Specialization
  - Creating new classes from existing classes proving more specialized attributes or operations
- Inheritance or Class Hierarchies
  - Organization of our inheritance relationships

Inheritance

Class hierarchy

Classes:

- A
- B is derived from A
- C is derived from A
- D is derived from C
- E is derived from D

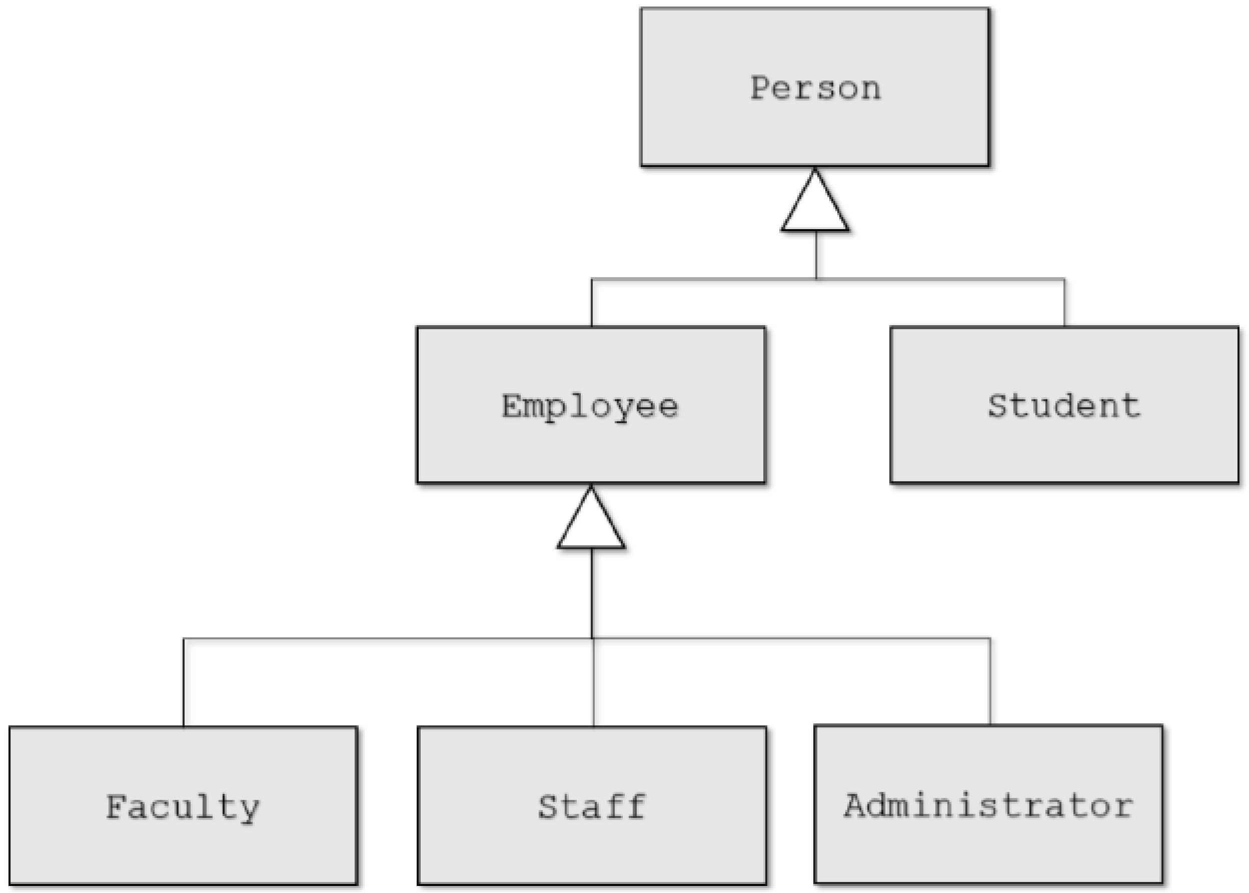


Inheritance

Class hierarchy

Classes:

- Person
- Employee is derived from Person
- Student is derived from Person
- Faculty is derived from Employee
- Staff is derived from Employee
- Administrator is derived from Employee



Composition

Inheritance

Public Inheritance vs. Composition

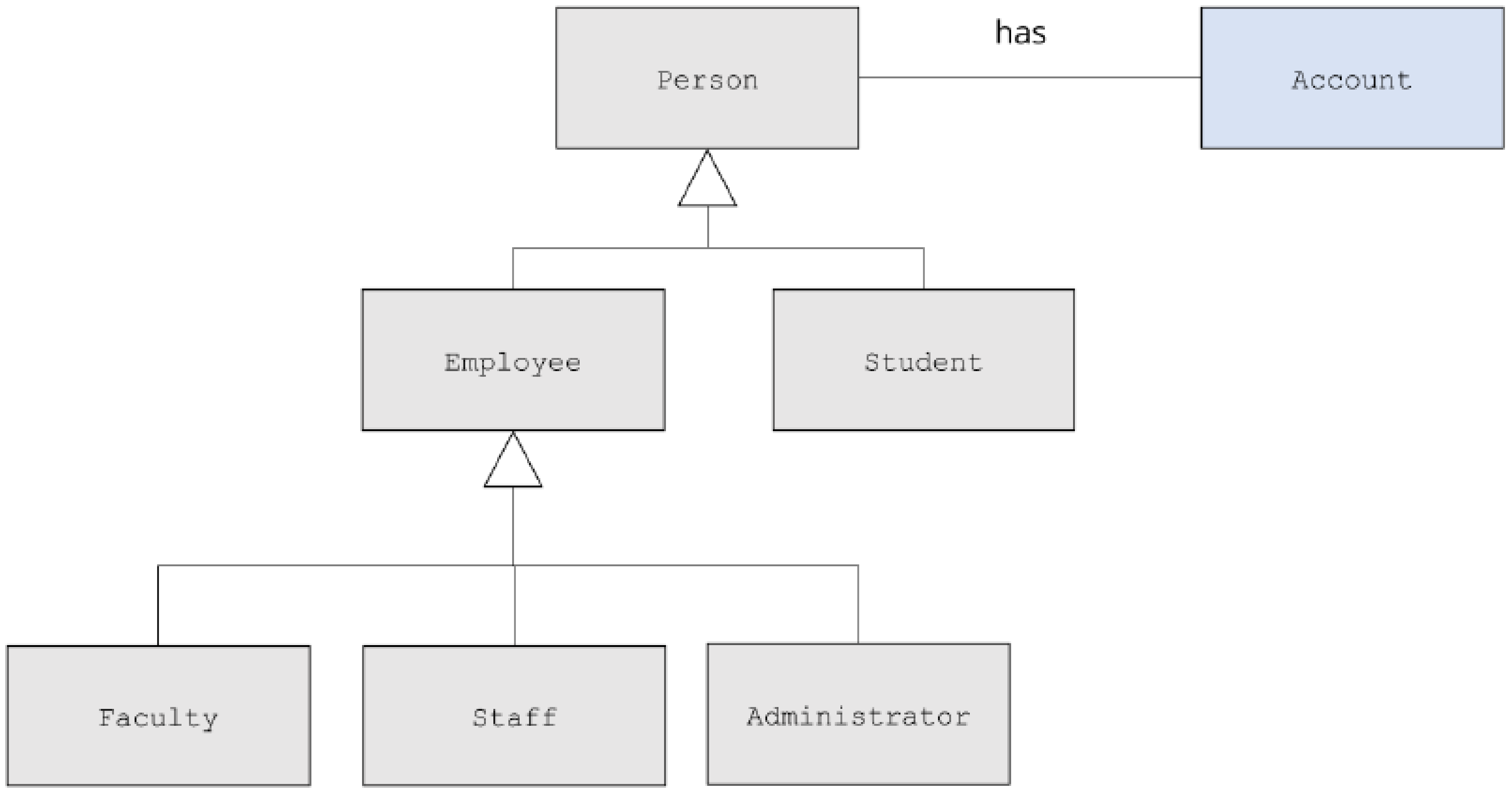
- Both allow reuse of existing classes

- Public Inheritance
  - “is-a” relationship
    - Employee ‘is-a’ Person
    - Checking Account ‘is-a’ Account
    - Circle “is-a” Shape

- Composition
  - “has-a” relationship
    - Person “has a” Account
    - Player “has-a” Special Attack
    - Circle “has-a” Location

Inheritance

Public Inheritance vs. Composition



Inheritance

Public Inheritance vs. Composition

```
class Person {
private:
   std::string name;  // has-a name
   Account account; // has-a account
};
```

Deriving Classes from Existing Classes

Deriving classes from exiting classes

C++ derivation syntax

```
class Base {
  // Base class members . . .
};

class Derived: access-specifier Base {
    // Derived class members . . .
};
```

Access-specifier can be: `public`, `private`, or `protected`

Deriving classes from exiting classes

Types of inheritance in C++

- `public`
  - Most common
  - Establishes '**is-a**' relationship between Derived and Base classes

- `private` and `protected`
  - Establishes "derived class **has a** base class" relationship
  - "Is implemented in terms of" relationship
  - Different from composition

Deriving classes from exiting classes

C++ derivation syntax

```
class Account {
  // Account class members . . .
};

class Savings_Account: public Account {
    // Savings_Account class members . . .
};
```

`Savings_Account` 'is-a' `Account`

Deriving classes from exiting classes

C++ creating objects

```
Account account {};
Account *p_account = new Account();

account.deposit(1000.0);
p_account->withdraw(200.0);

delete p_account;
```

Deriving classes from exiting classes

C++ creating objects

```
Savings_Account sav_account {};
Savings_Account *p_sav_account = new Savings_Account();

sav_account.deposit(1000.0);
p_sav_account->withdraw(200.0);

delete p_sav_account;
```

Protected Members and Class Access

Protected Members and Class Access

The protected class member modifier

```
class Base {

  protected:
    // protected Base class members . . .
};
```

•Accessible from the Base class itself
•Accessible from classes Derived from Base
•Not accessible by objects of Base or Derived

Protected Members and Class Access

The protected class member modifier

```
class Base {
   public:
     int a;   // public Base class members . . .


  protected:
     int b; // protected Base class members . . .


  private:
     int c; // private Base class members . . .

};
```

Deriving classes from exiting classes

Access with **`public`** inheritance

Base Class

```
public: a
protected: b
private: c
```

public
inheritance

Access in
Derived Class

```
public: a
protected: b
c : no access
```

Deriving classes from exiting classes

Access with **`protected`** inheritance

Base Class

```
public: a
protected: b
private: c
```

protected
inheritance

Access in
Derived Class

```
protected: a
protected: b
c : no access
```

Deriving classes from exiting classes

Access with **`private`** inheritance

Base Class

```
public: a
protected: b
private: c
```

private
inheritance

Access in
Derived Class

```
private: a
private: b
c : no access
```

Access Modifiers Vs Visibility Modifiers
More details on Class Access

Public, Private and Protected Access Level

- Public Data Members and Member Functions:

  - Access Level: Public members are accessible from anywhere, including outside the class.
  - Accessible From:
    - Other classes.
    - Objects of the class.
    - Functions or code outside the class.

Public, Private and Protected Access Level

```
class MyClass {
public:
    int publicVar; // Public data member
    void publicFunction() {
        // Public member function
    }
};
int main() {
    MyClass obj;
    obj.publicVar = 42;      // Accessing public data member
    obj.publicFunction();   // Accessing public member function
    return 0;
}
```

Public, Private and Protected Access Level

- Protected Data Members and Member Functions:
  - Access Level: Protected members are accessible within the class itself and by derived classes (subclasses).
  - Not Accessible From:
    - Code outside the class hierarchy (unless using a friend function or method).

Public, Private and Protected Access Level

```
class MyBaseClass {
protected:
    int protectedVar; // Protected data member
    void protectedFunction() {
        // Protected member function
    }
};
class MyDerivedClass : public MyBaseClass {
public:
    void someFunction() {
        protectedVar = 42;       // Accessing protected data member from derived class
        protectedFunction();    // Accessing protected member function from derived class
    }
};
```

Public, Private and Protected Access Level

- Private Data Members and Member Functions:

- Access Level: Private members are only accessible within the class itself.
- Not Accessible From:
  - Derived classes.
  - Code outside the class.

Public, Private and Protected Access Level

```
class MyClass {
private:
    int privateVar; // Private data member
    void privateFunction() {
        // Private member function
    }
public:
    void somePublicFunction() {
        privateVar = 42;        // Accessing private data member from within the class
        privateFunction();     // Accessing private member function from within the class
    }
};
```

Public, Private and Protected Access Level

```
int main() {
    MyClass obj;
    // obj.privateVar = 42;      // Error: Cannot access private data member from outside the class
    // obj.privateFunction();  // Error: Cannot access private member function from outside the class
    obj.somePublicFunction(); // Accessing public member function
    return 0;
}
```

Public, Private and Protected Access Level

- Public members are accessible from anywhere.
- Protected members are accessible within the class and by derived classes.
- Private members are only accessible within the class itself.

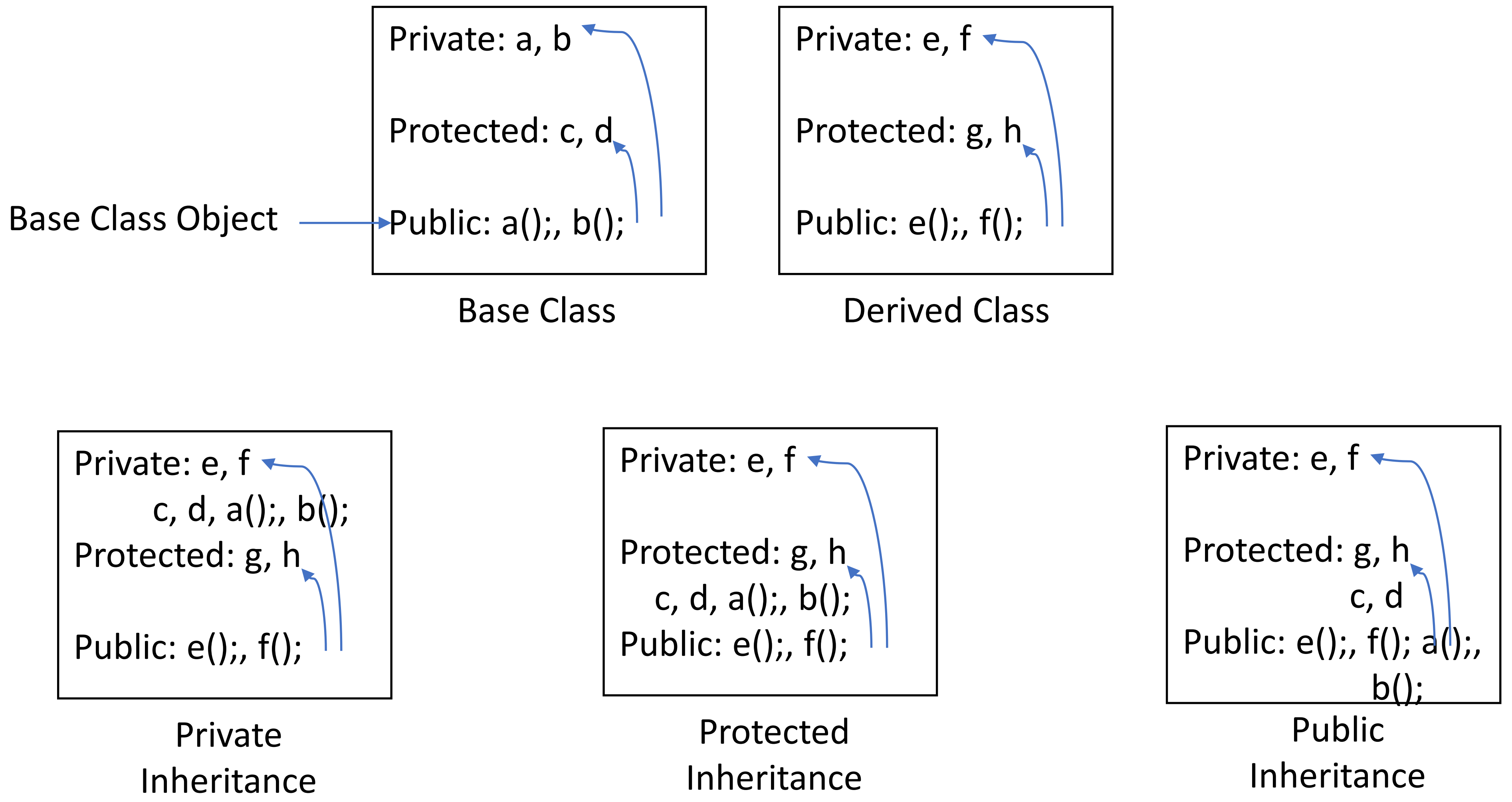
Also Remember that:

- Private and protected members can only be accessed using public classes

Access Modifiers vs Visibility Modifiers

Visibility Modifiers	Data Members/ Member Functions		
	Private	Protected	Public
Private	Nil	Private	Private
Protected	Nil	Protected	Protected
Public	Nil	Protected	Public

Public, Private and Protected Access Level



Constructors and Destructors

Constructors and Destructors

Constructors and class initialization

- A Derived class inherits from its Base class

- The Base part of the Derived class MUST be initialized BEFORE the Derived class is initialized

- When a Derived object is created
  - Base class constructor executes then
  - Derived class constructor executes

Constructors and Destructors

Constructors and class initialization

```
class Base {
public:
   Base(){ cout << "Base constructor" << endl; }
};

class Derived : public Base {
public:
   Derived(){ cout << "Derived constructor " << endl; }
};
```

Constructors and Destructors

Constructors and class initialization

Output

```
Base base;
```

```
Base constructor
```

```
Derived derived;
```

```
Base constructor
Derived constructor
```

Constructors and Destructors

Destructors

- Class destructors are invoked in the reverse order as constructors

- The Derived part of the Derived class MUST be destroyed BEFORE the Base class destructor is invoked

- When a Derived object is destroyed
  - Derived class destructor executes then
  - Base class destructor executes
  - Each destructor should free resources allocated in it’s own constructors

Constructors and Destructors

Destructors

```
class Base {
public:
   Base(){ cout << "Base constructor" << endl; }
   ~Base(){ cout << "Base destructor" << endl; }
};

class Derived : public Base {
public:
   Derived(){ cout << "Derived constructor " << endl; }
   ~Derived(){ cout << "Derived destructor " << endl; }
};
```

Constructors and Destructors

Destructors and class initialization

Output

```
Base base;
```

```
Base constructor
Base destructor
```

```
Derived derived;
```

```
Base constructor
Derived constructor
Derived destructor
Base destructor
```

Constructors and Destructors

Constructors and class initialization

- A Derived class does NOT inherit
  - Base class constructors
  - Base class destructor
  - Base class overloaded assignment operators
  - Base class friend functions

- However, the derived class constructors, destructors, and overloaded assignment operators can invoke the base-class versions

- C++11 allows explicit inheritance of base 'non-special' constructors with
  - `using Base::Base;` anywhere in the derived class declaration
  - Lots of rules involved, often better to define constructors yourself

Passing Arguments to Base Class Constructors

Inheritance

Passing arguments to base class constructors

- The Base part of a Derived class must be initialized first

- How can we control exactly which Base class constructor is used during initialization?

- We can invoke the whichever Base class constructor we wish in the initialization list of the Derived class

Inheritance

Passing arguments to base class constructors

```
class Base {
public:
   Base();
   Base(int);
   . . .
};

Derived::Derived(int x)
   : Base(x), {optional initializers for Derived} {
  // code
}
```

Constructors and Destructors

Constructors and class initialization

```
class Base {
   int value;
public:
   Base(): value{0} {
      cout << "Base no-args constructor" << endl;
   }
   Base(int x) : value{x} {
      cout << "int Base constructor" << endl;
   }
};
```

Constructors and Destructors

Constructors and class initialization

```
class Derived : public Base {
   int doubled_value;
public:
   Derived(): Base{}, doubled_value{0} {
      cout << "Derived no-args constructor " << endl;
   }
   Derived(int x) : Base{x}, doubled_value {x*2} {
      cout << "int Derived constructor " << endl;
   }
};
```

Constructors and Destructors

Constructors and class initialization

Output

```
Base base;                    Base no-args constructor

Base base{100};               int Base constructor

Derived derived;              Base no-args constructor
                              Derived no-args constructor

Derived derived{100};         int Base constructor
                              int Derived constructor
```

Copy/Move Constructors and Operator = with Derived Classes

Inheritance

Copy/Move constructors and overloaded operator=

- Not inherited from the Base class

- You may not need to provide your own
  - Compiler-provided versions may be just fine

- We can explicitly invoke the Base class versions from the Derived class

Inheritance

Copy constructor

- Can invoke Base copy constructor explicitly
  - Derived object '*other*' will be **sliced**

```
Derived::Derived(const Derived &other)
    : Base(other), {Derived initialization list}
{
    // code
}
```

Constructors and Destructors

Copy Constructors

```
class Base {
    int value;
public:
    // Same constructors as previous example

    Base(const Base &other) :value{other.value} {
  cout << "Base copy constructor" << endl;
    }
};
```

Constructors and Destructors

Copy Constructors

```
class Derived : public Base {
   int doubled_value;
public:
   // Same constructors as previous example

   Derived(const Derived &other)
  : Base(other), doubled_value {other.doubled_value } {
         cout << "Derived copy constructor " << endl;
   }
};
```

Constructors and Destructors

operator=

```
class Base {
   int value;
public:
   // Same constructors as previous example
   Base &operator=(const Base &rhs) {
      if (this != &rhs) {
         value = rhs.value; // assign
      }
      return *this;
   }
};
```

Constructors and Destructors

operator=

```
class Derived : public Base {
   int doubled_value;
public:
   // Same constructors as previous example
   Derived &operator=(const Derived &rhs) {
      if (this != &rhs) {
         Base::operator=(rhs);    // Assign Base part
         doubled_value = rhs.doubled_value; // Assign Derived part
      }
      return *this;
   }
};
```

Redefining Base Class Methods

Inheritance

Copy/Move constructors and overloaded operator=

- Often you do not need to provide your own

- If you **DO NOT** not define them in Derived
  - then the compiler will create them and automatically and call the base class's version

- If you **DO** provide Derived versions
  - then **YOU** must invoke the Base versions **explicitly** yourself

- Be careful with raw pointers
  - Especially if Base and Derived each have raw pointers
  - Provide them with deep copy semantics

Inheritance

Using and redefining Base class methods

- Derived class can directly invoke Base class methods
- Derived class can **override** or **redefine** Base class methods
- Very powerful in the context of polymorphism
(next section)

Inheritance

Using and redefining Base class methods

```
class Account {
public:
   void deposit(double amount) { balance += amount; }
};

class Savings_Account: public Account {
public:
   void deposit(double amount) { // Redefine Base class method
      amount += some_interest;
      Account::deposit(amount);  // invoke call Base class method
   }
};
```

Inheritance

Static binding of method calls

- Binding of which method to use is done at compile time
  - Default binding for C++ is static
  - Derived class objects will use Derived::deposit
  - But, we can explicitly invoke Base::deposit from Derived::deposit
  - OK, but limited – much more powerful approach is dynamic binding which we will see in the next section

Inheritance

Static binding of method calls

```
Base b;
b.deposit(1000.0);        // Base::deposit


Derived d;
d.deposit(1000.0);        // Derived::deposit


Base *ptr = new Derived();
ptr->deposit(1000.0);      // Base::deposit ????
```

Types of Inheritance

Single Inheritance

- Single inheritance is a fundamental concept in object-oriented programming (OOP) and represents a relationship between classes in which one class, known as the derived class or subclass, inherits the attributes and behaviors (members) of a single base class or superclass. In single inheritance:

  Base Class (Superclass): This is the class whose members are inherited by the derived class. It serves as the foundation for the derived class.

  Derived Class (Subclass): This is the class that inherits the members of the base class. It can add additional members or override the inherited members.

Single Inheritance (Important concepts)

1. Inheritance: Single inheritance is a form of inheritance, which is one of the four fundamental OOP principles (Encapsulation, Abstraction, Inheritance, and Polymorphism). Inheritance allows for code reuse and the creation of class hierarchies.

2. Reusability: Single inheritance promotes the reuse of code. The derived class can inherit and use the attributes and methods of the base class, reducing the need to rewrite common functionality.

3. Code Extensibility: While the derived class inherits the members of the base class, it can also extend or enhance the functionality. This is achieved by adding new attributes and methods specific to the derived class.

4. Method Overriding: In single inheritance, the derived class can override (replace) the inherited methods from the base class. This allows the derived class to provide its own implementation for specific behaviors.

Single Inheritance (Important concepts)

5. Access Control: Access control specifiers (public, private, protected) determine the visibility of inherited members in the derived class. Typically, public members of the base class remain public in the derived class, while private members remain inaccessible. Protected members are often inherited as protected in the derived class.

6. Base Class and Derived Class Relationship: The relationship between the base class and the derived class is "is-a." This means that the derived class "is a kind of" the base class. For example, if you have a base class "Vehicle" and a derived class "Car," you can say that a car is a kind of vehicle.

7. Code Organization: Single inheritance helps organize code and promotes a clear class hierarchy. This can improve code maintainability and understandability.

8. Multiple Inheritance: Single inheritance is distinct from multiple inheritance, where a class can inherit from multiple base classes. In single inheritance, a class inherits from only one base class.

Single Inheritance (Important concepts)

Single inheritance is a powerful concept in OOP, and it allows for the creation of structured and organized class hierarchies, making code more modular, reusable, and easier to maintain. It is a fundamental building block in the design of object-oriented systems.

Multiple Inheritance

Multiple inheritance is a concept in object-oriented programming that allows a class to inherit attributes and behaviors from more than one base class. In other words, a class in a programming language that supports multiple inheritance can inherit members (such as attributes and methods) from multiple parent classes. In multiple inheritance:

Base Class: A base class is a class whose members can be inherited by other classes. In multiple inheritance, a class can have multiple base classes.

Derived Class: A derived class is a class that inherits members from one or more base classes. It combines the attributes and behaviors of its parent classes.

Multiple Inheritance (Important concepts)

1. Ambiguity: One of the key challenges in multiple inheritance is dealing with ambiguity when a member with the same name is inherited from multiple base classes. Language-specific rules or mechanisms are used to resolve such ambiguities.

2. Diamond Problem: The diamond problem is a specific issue that can arise in multiple inheritance when a class inherits from two classes that have a common base class. This can result in ambiguity, and it's a challenge that languages must address.

3. Virtual Inheritance: Some programming languages, like C++ and Python, support virtual inheritance to address the diamond problem. In virtual inheritance, only one instance of the common base class is shared among the derived classes.

4. Order of Inheritance: The order in which base classes are listed in the declaration of a derived class can impact the behavior of the derived class. This is particularly important in languages like C++.

Multiple Inheritance (Important concepts)

5. Mixin Classes: Mixin classes are a common use of multiple inheritance. These are classes that provide a specific set of methods or attributes that can be reused in multiple classes without having to reimplement the same functionality.

6. Interface Inheritance: In some languages, multiple inheritance is used to implement interfaces, which define a set of methods that a class must implement. By inheriting from multiple interfaces, a class can provide various sets of functionality.

7. Complexity: Multiple inheritance can lead to complex class hierarchies, which can make code harder to understand and maintain. It's essential to use multiple inheritance judiciously and design class hierarchies carefully.

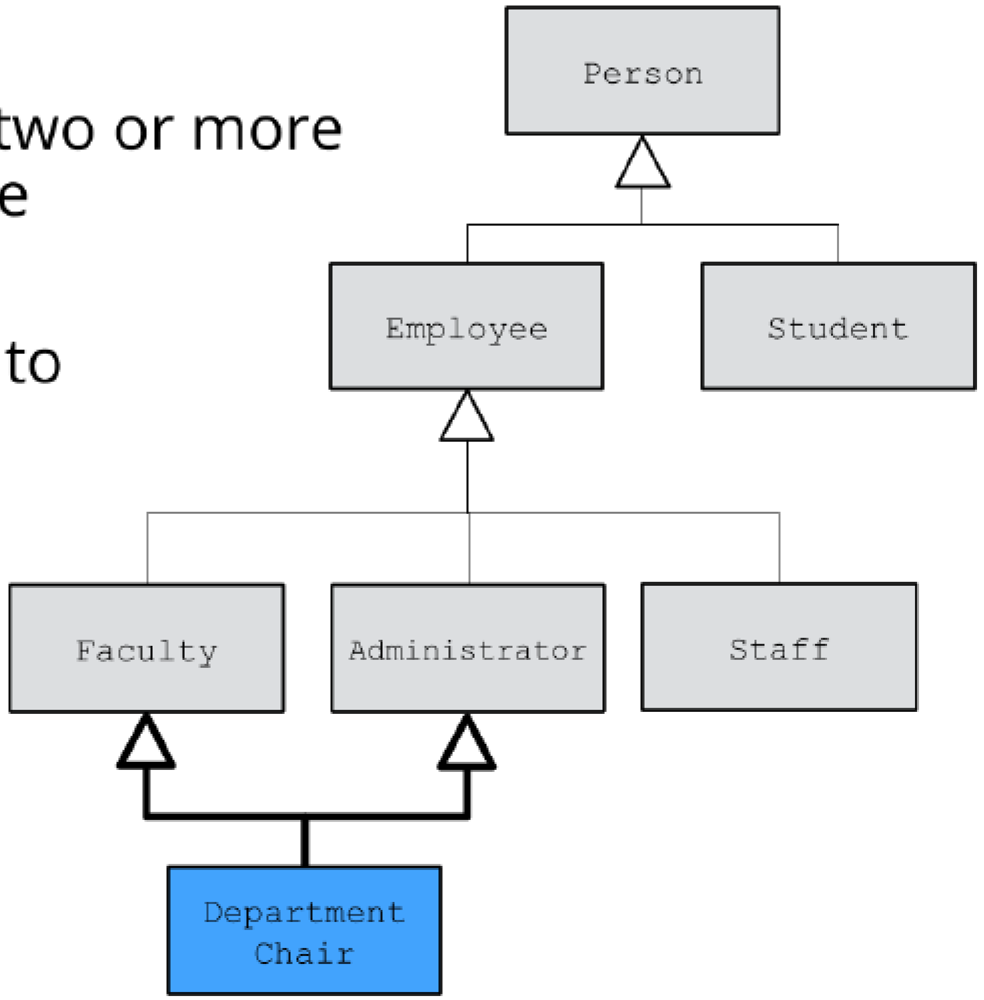
8. Language Support: Not all programming languages support multiple inheritance. Some languages, like Java, support only single inheritance to avoid the complexities and ambiguities associated with multiple inheritance. Others, like C++, support multiple inheritance with certain rules and features to address potential issues.

Multiple Inheritance (Important concepts)

In practice, multiple inheritance can be a powerful tool for code reuse and design, but it should be used with caution, and careful design considerations are necessary to avoid ambiguity and complex class hierarchies.

Multiple Inheritance

- A derived class inherits from two or more Base classes at the same time
- The Base classes may belong to unrelated class hierarchies
- A Department Chair
  - Is-A Faculty and
  - Is-A Administrator



Multiple Inheritance

C++ Syntax

```
class Department_Chair:
    public Faculty, public Administrator {
  . . .
};
```

- Some compelling use-cases
- Easily misused
- Can be very complex

Multilevel Inheritance

Multilevel inheritance is a type of inheritance in object-oriented programming where a class inherits from another class, which in turn may inherit from another class, forming a chain of inheritance. In multilevel inheritance, a class at one level serves as the base class for a derived class at the next level, and this can continue through multiple levels. In multilevel inheritance:

Base Class: The base class is the class that serves as the parent class, and it provides attributes and methods that can be inherited by derived classes. In multilevel inheritance, there can be multiple levels of base classes.

Derived Class: The derived class is a class that inherits attributes and methods from a base class. In multilevel inheritance, a derived class at one level can serve as the base class for another class at the next level.

Multilevel Inheritance (Important concepts)

1. Chain of Inheritance: Multilevel inheritance forms a chain of inheritance where a class inherits from another class, and this can continue through several levels. Each class inherits attributes and behaviors from its immediate parent class in the chain.

2. Inheritance Hierarchy: In a multilevel inheritance hierarchy, you can have multiple classes arranged in a hierarchical structure, with each class inheriting from the one above it. This hierarchy allows for the organization and sharing of code.

3. Access Control: In multilevel inheritance, access control modifiers like public, protected, and private play an essential role in determining how the attributes and methods of the base class are inherited and accessed in the derived class.

4. Method Overriding: Derived classes in a multilevel inheritance hierarchy can override the methods of their parent classes to provide their own implementation. This allows for customization of behavior at each level.

Multilevel Inheritance (Important concepts)

5. Diamond Problem: Multilevel inheritance can lead to the "diamond problem," which occurs when a class inherits from two or more classes that have a common base class. This can lead to ambiguities in method and attribute access. To address this, some programming languages provide features like virtual inheritance.

6. Code Reusability: Multilevel inheritance promotes code reusability by allowing you to create a hierarchy of related classes. Common attributes and behaviors can be defined in base classes and reused in derived classes.

Hierarchical Inheritance

Hierarchical Inheritance is one of the common types of inheritance in object-oriented programming. It involves the creation of a class hierarchy where multiple derived classes inherit from a single base class. In hierarchical inheritance, the base class serves as a common ancestor for several derived classes, forming a tree-like structure. Each derived class inherits the attributes and behaviors of the base class while adding its unique attributes and behaviors. This type of inheritance is valuable for modeling relationships in scenarios where multiple classes share a common set of characteristics. In hierarchical inheritance:

Base Class (Superclass): The base class is also known as the superclass. It defines the common attributes and methods that are shared by all derived classes.

Derived Classes (Subclasses): These are the classes that inherit from the base class. Each derived class can have its attributes and methods in addition to the ones inherited from the base class.

Hierarchical Inheritance (Important concepts)

1. Inheritance: Hierarchical inheritance involves the concept of inheritance, where derived classes inherit the attributes and methods of the base class. This allows code reusability and the sharing of common characteristics.

2. Code Reusability: One of the primary benefits of hierarchical inheritance is code reusability. The common attributes and behaviors defined in the base class need not be re-implemented in every derived class.

3. Method Overriding: Derived classes can provide their implementations of methods inherited from the base class. This process is known as method overriding and allows customization of behavior in each derived class.

4. Tree Structure: Hierarchical inheritance forms a tree-like structure, where the base class is at the root, and multiple derived classes branch out from it. Each derived class may have its further subclasses.

Hierarchical Inheritance (Important concepts)

5. Polymorphism: Inheritance in general, including hierarchical inheritance, facilitates the use of polymorphism, where objects of derived classes can be treated as objects of the base class. This allows for more flexible and generic coding.

6. Superclass Methods: The methods of the base class can be accessed and called by instances of derived classes. This is essential for invoking common behaviors shared among multiple classes.

Hybrid Inheritance (Important Concepts)

Hybrid inheritance is a combination of different types of inheritance within a programming language. It often combines two or more inheritance mechanisms, typically single inheritance and multiple inheritance, to achieve a more complex class hierarchy. Hybrid inheritance allows for a richer and more flexible organization of classes and relationships between them.

1. Single Inheritance: In single inheritance, a derived class inherits from a single base class. This is the simplest form of inheritance.

2. Multiple Inheritance: In multiple inheritance, a derived class can inherit from multiple base classes. This allows a class to inherit attributes and behaviors from more than one parent class.

3. Virtual Base Classes: In multiple inheritance scenarios, the use of virtual base classes helps prevent issues like the "diamond problem." When a class virtually inherits a base class, there is only one instance of that base class in the class hierarchy, avoiding ambiguity.

Hybrid Inheritance (Important Concepts)

4. Diamond Problem: The diamond problem occurs in multiple inheritance when a class inherits from two classes that have a common base class. This can create ambiguity in the class hierarchy. Virtual inheritance is often used to resolve this problem.

5. Order of Constructors and Destructors: In hybrid inheritance, the order of constructor and destructor calls is critical to ensure proper initialization and cleanup of objects. This order can be complex, especially in multiple inheritance scenarios.

6. Mixins and Interfaces: In some programming languages, hybrid inheritance can be implemented using mixins or interfaces. Mixins are classes that provide a specific set of features, and a class can inherit from multiple mixins to combine these features.

7. Method Overriding: In hybrid inheritance, method overriding becomes crucial as derived classes may need to override methods from multiple base classes.

Hybrid Inheritance (Important Concepts)

8. Code Reusability: One of the main advantages of hybrid inheritance is code reusability. By inheriting from multiple base classes, a derived class can reuse code from different sources.

9. Complexity and Ambiguity: Hybrid inheritance can introduce complexity and potential ambiguity into the class hierarchy. Careful design and consideration of the class structure are necessary to avoid issues.

10. Object-Oriented Design: Proper object-oriented design principles, such as the SOLID principles and design patterns, are important when working with hybrid inheritance to maintain a clear and maintainable codebase.

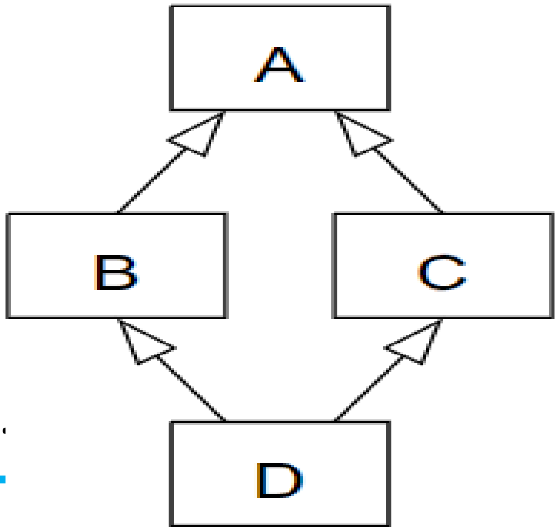
Hybrid inheritance, while powerful, should be used with caution, and developers should be aware of the potential challenges and complexities it can introduce. Proper design and documentation are crucial to ensure that the class hierarchy is both functional and understandable.

Diamond Problem

The diamond problem occurs in multiple inheritance when a class inherits from two classes that have a common base class. This can create ambiguity in the class hierarchy. Virtual inheritance is often used to resolve this problem.

In C++, virtual functions can be used to resolve the diamond problem caused by multiple inheritance. Virtual functions enable dynamic binding, which allows the program to determine at runtime which version of the function to call based on the actual object type. When you use virtual inheritance, it ensures that there is only one instance of a base class in the hierarchy, which effectively resolves the ambiguity that arises in the diamond problem.

Without virtual, the copy of the functionA (say it is a public function in base class A) in base class will inherit to class B, and class C. when class D inherited from B & C, it will contain two copies of contents from A. This would result in multiple copies of functionA in the derived class D, leading to ambiguity and potential issues when you try to call or access the members of the base class A.


$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Diamond Problem

The diamond problem arises because class D inherits from both B and C, both of which inherit from A. In this scenario, the ambiguity occurs when you try to call a method defined in A, as it's not clear which version of the method should be invoked.

To address the diamond problem, you have used virtual inheritance on class A. However, you haven't used virtual inheritance on classes B and C. Virtual inheritance should be used consistently throughout the hierarchy to ensure that there is only one shared instance of the virtual base class A.

Diamond Problem (Solution: Virtual Inheritance) (Example 1)

Derived Classes Implement the Virtual Function:

Create the derived classes, which inherit virtually from the base class. These derived classes must provide concrete implementations of the virtual function.

```
class Derived1 : virtual public Base {
public:
    void display() override {
        std::cout << "Derived1" << std::endl;
    }
};

class Derived2 : virtual public Base {
public:
    void display() override {
        std::cout << "Derived2" << std::endl;
    }
};
```

Diamond Problem

Resolving the Diamond Problem:

Create a class that inherits from the derived classes that have virtual inheritance. This class doesn't need to provide an implementation of the virtual function; it inherits the implementations from the derived classes.

```
class ResolvedDiamond : public Derived1, public Derived2 {
};
```

Now, when you create an object of the ResolvedDiamond class and call the display function on it, the program can determine which version of the function to call based on the actual object's type. This resolves the diamond problem by providing a clear and unambiguous path for calling the virtual functions.

```
ResolvedDiamond obj;
obj.display();   // Calls either Derived1::display or Derived2::display based on the the actual object's type
```

Diamond Problem (Example 2)

Resolving the Diamond Problem:

class A { public: void Foo() {} }
class B : public
virtual A {}
class C : public
virtual A {}
class D : public B, public C {}

class B: virtual A means , that any class inherited from B is now responsible for creating A by itself, since B isn't going to do it automatically.

D d
d.Foo
(); // no longer ambiguous

Method Overriding

Definition: Method overriding is a concept in object-oriented programming (OOP) where a derived class provides a specific implementation for a method that is already defined in its base class. In C++, method overriding is achieved using virtual functions and the override keyword.

Difference b/w Method Redefinition and Method Overriding

Method Overriding:

1. **Inheritance Requirement:**
   - Overriding occurs in the context of inheritance. It involves a base class declaring a virtual method, and a derived class providing a specific implementation for that method.
2. **Use of virtual Keyword:**
   - The base class method that is intended to be overridden is declared as **virtual**. This keyword signals to the compiler that the method can be overridden by derived classes.
3. **override Keyword:**
   - In the derived class, the **override** keyword is used explicitly to indicate the intention to override a virtual method from the base class. This helps catch potential errors during compilation if there is a mismatch between the base class and derived class methods.
4. **Dynamic Binding:**
   - Overridden methods support dynamic binding, enabling polymorphic behavior. The appropriate method is determined at runtime based on the actual type of the object.

Difference b/w Method Redefinition and Method Overriding

Method Redefinition:

1. **Independent Methods:**
   - Redefinition is a broader term and can refer to any situation where a derived class provides a new definition for a method, whether or not the method is virtual or declared in a base class.
2. **Not Limited to Inheritance:**
   - Redefinition is not necessarily tied to inheritance. A class can redefine a method without the presence of a virtual function or a base class.
3. **No virtual or override Keywords:**
   - Unlike method overriding, redefinition does not involve the use of the virtual or override keywords. Redefined methods are simply new implementations of existing methods.
4. **Static Binding:**
   - Redefined methods typically result in static binding, where the method to be called is determined at compile-time based on the declared type of the object.

Ambiguity Resolution in C++

Ambiguity in C++ arises when the compiler encounters a situation where it is unable to determine the correct method or variable to use due to multiple valid options. This can occur in cases of multiple inheritance, function overloading, or template specialization.

Relationships

Relationships

OOP generally support following types of relationships

1. Inheritance/ is a relationship/ Generalization
2. Association
3. Aggregation
4. Composition

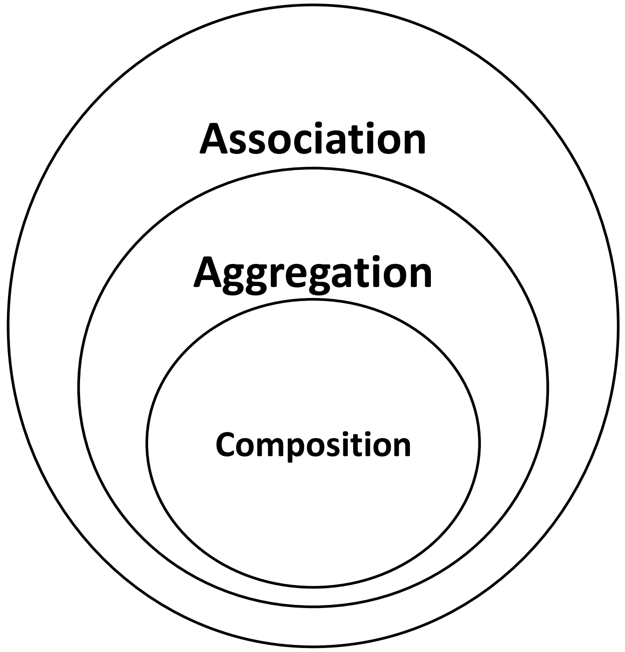
- All the relationships is based on ‘is-a’, ‘has-a’ and ‘part-of’ relationships.
- Generalization is also called as ‘is-a’ relationship. At a broader level ’is-a’ relationship is same as inheritance.

Multiplicity

- 0 → None
- 1 → Exactly one
- * → Many
- 1+ → one or more
- 0..1 → None or one
- 3..6 → 3,4,5,6
- 7,8 → 7 or 8

Association (has a relationship)

- It is a relationship between two separate classes which establishes through their objects
- Association can be one-to-one, one-to-many, many-to-one and many-to-many
- In OOP, an object communicate to other object to use functionality and services provided by that object.
- Association is shown between classes.
- All the links are associations connecting objects.
- It can be read bidirectionally
- Represented by a line between classes
- Association is between classes and link is between objects
- Objects are independent. (Doctor, Patient)
- Composition and Aggregation are two forms of association



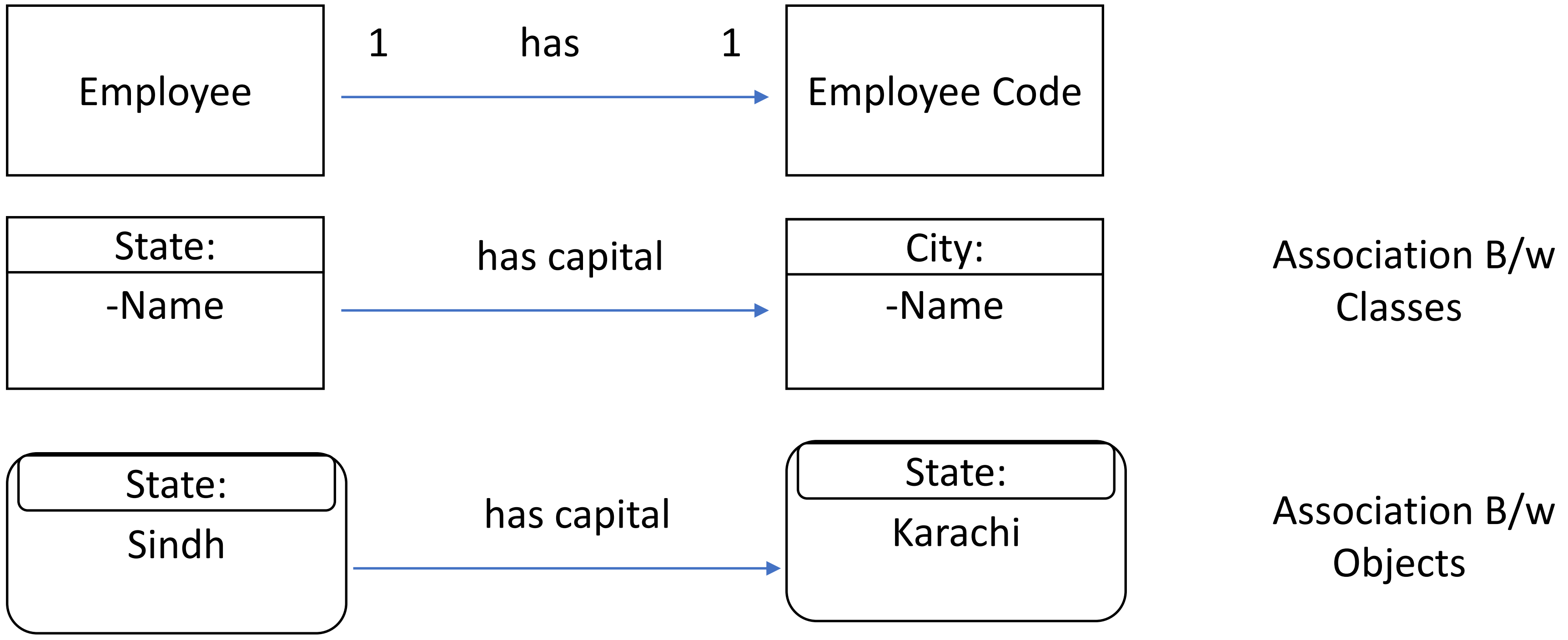
Association (has a relationship)

Association Types

- 1-1
- 1-many
- Many-1
- Many-many
- Ternary

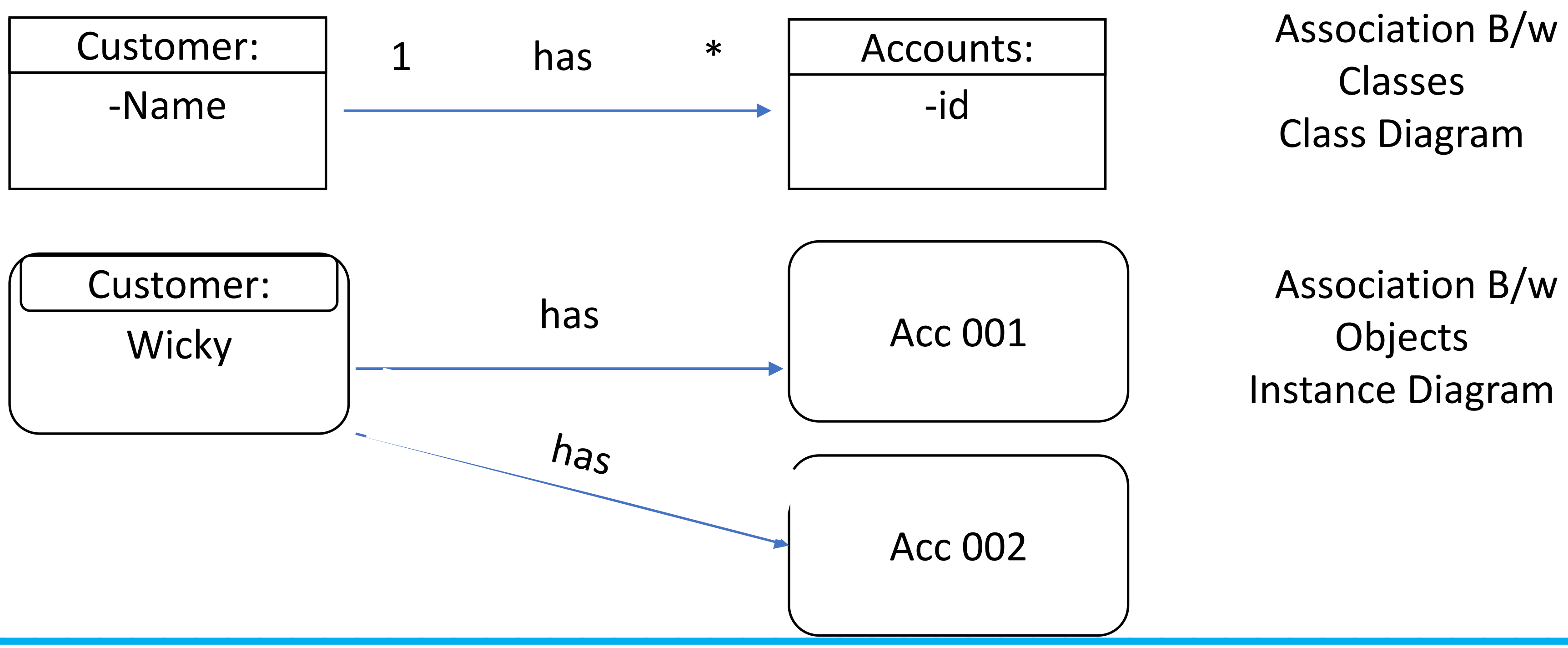
Association

- In 1-1 association only one object is associated with other object at a time
- The multiplicity symbol “1” is written at the of the link to represent that there is only one obj present in communication



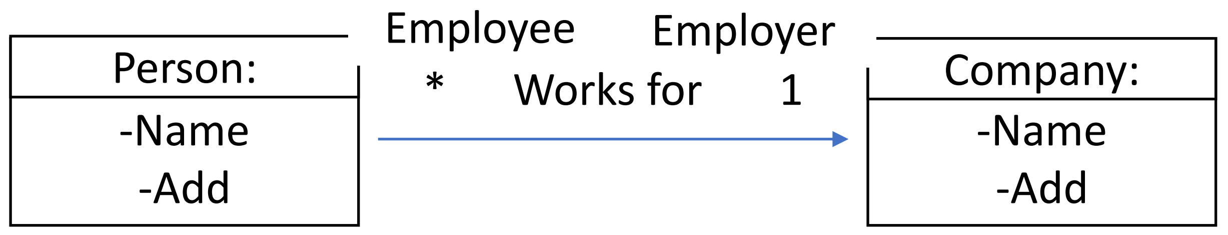
Association

- In 1-* association one object is associated with many object at a time
- The multiplicity symbol "*" is written at the of the link to represent that one obj present in communication with many objects at a time.

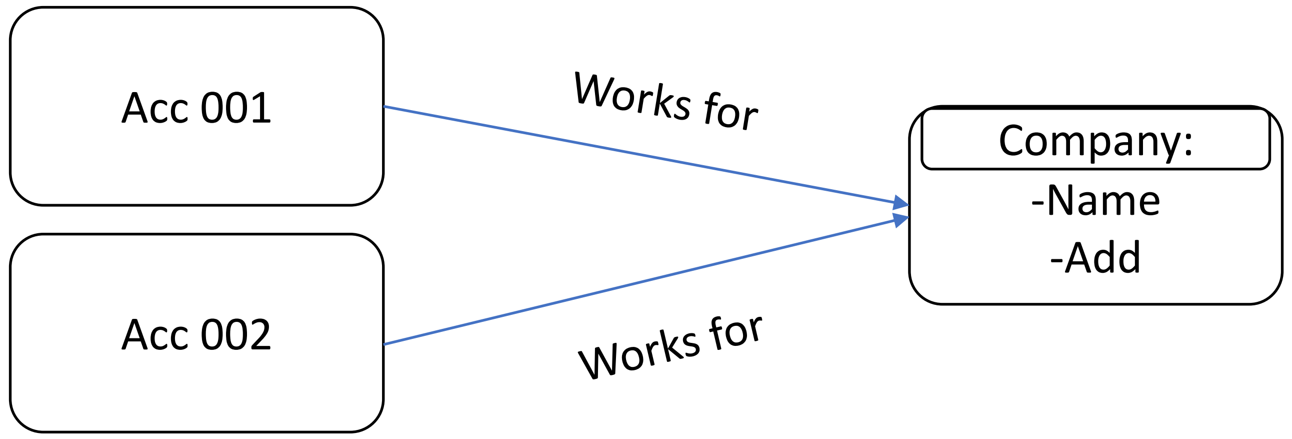


Association

- In *-1 association multiple objects are associated with one object at a time



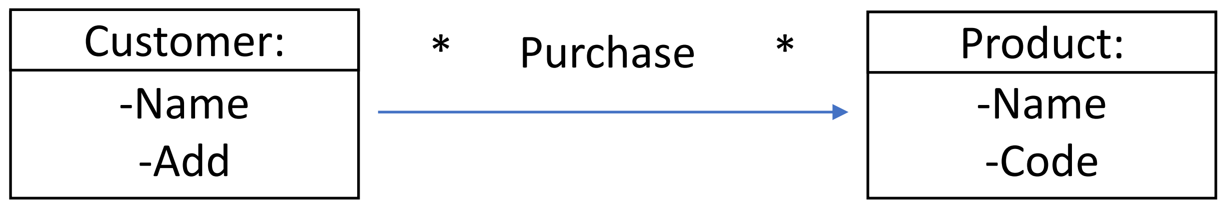
Association B/w Classes



Association B/w Objects

Association

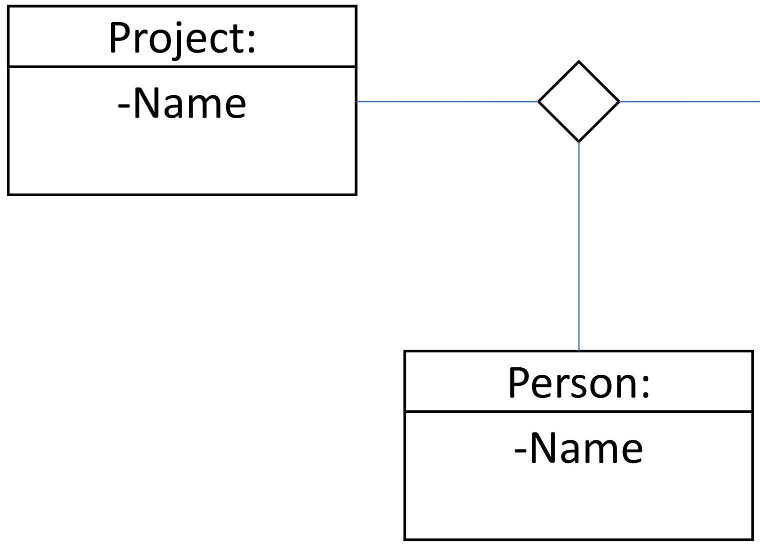
- In *-* association multiple objects are associated with multiple object at a time



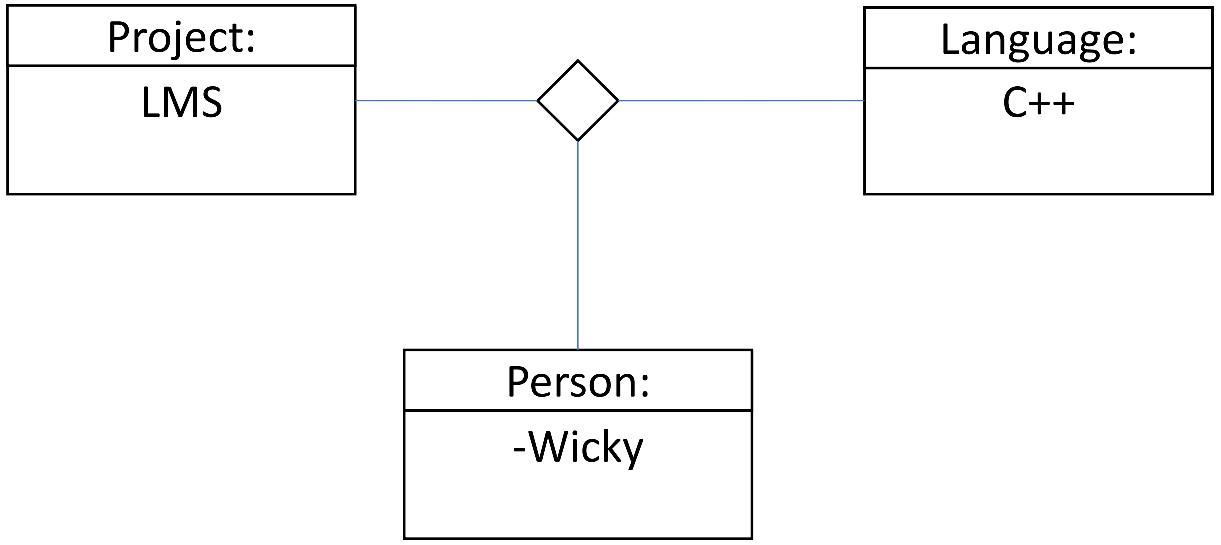
Association B/w
Classes

Ternary Association

- A ternary association describes a fact that involves three classes and cannot be split up into component binary association without loosing information
- A diamond shape is used to represent the ternary relationship



Association B/w Classes



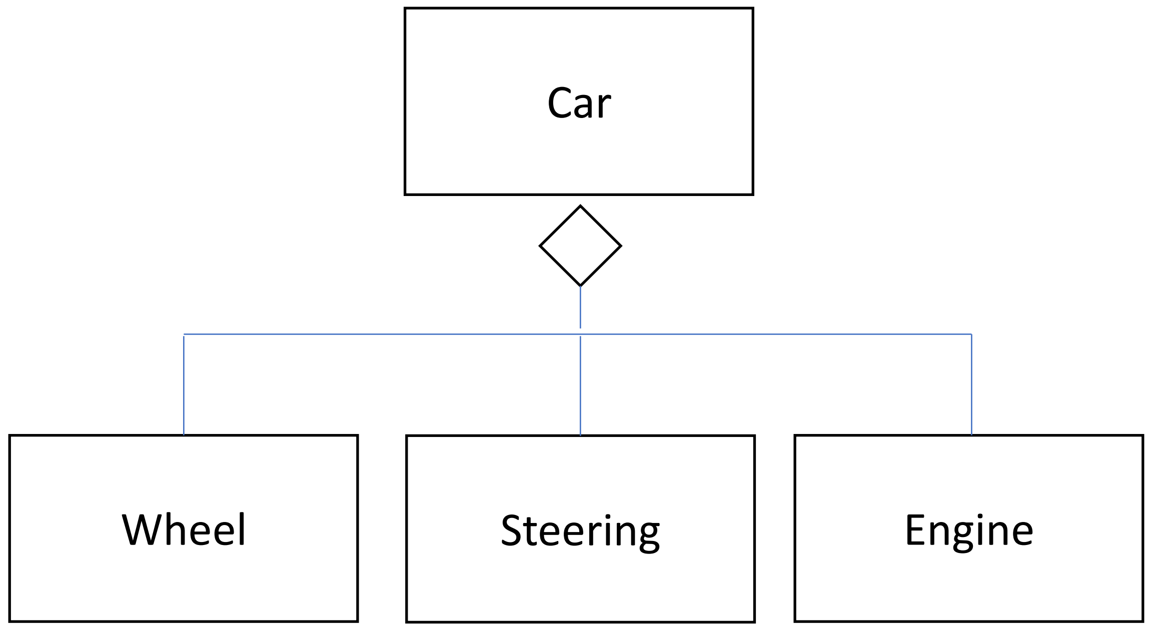
Association B/w Objects

Aggregation (part-of or part-whole relationship)

- An aggregation is a strong form of association, in which an aggregation object is made of components.
- Assembly part has diamond symbol connected with it.

Types

- Fixe Aggregation
- Variable Aggregation
- Recursive Aggregation



Aggregation
B/w Classes

Aggregation (part-of or part-whole relationship)

Fixed

- The number and types of the components are fixed or predefined.
- A car is a fixed aggregation of one engine, 4 wheels and one steering.

Variable

- Type of aggregation is fixed, but the number of parts of components are variable
- Company is variable aggregation of divisions and departments. Number of division and departments can vary.

Recursive Aggregation

- The number of levels are unlimited
- In recursive aggregation, an aggregate contains the components of its own types.
- A computer program is an aggregate of blocks with optimally recursive compound statements, the recursion terminates with simple statements.

Aggregation (part-of or part-whole relationship)

Properties

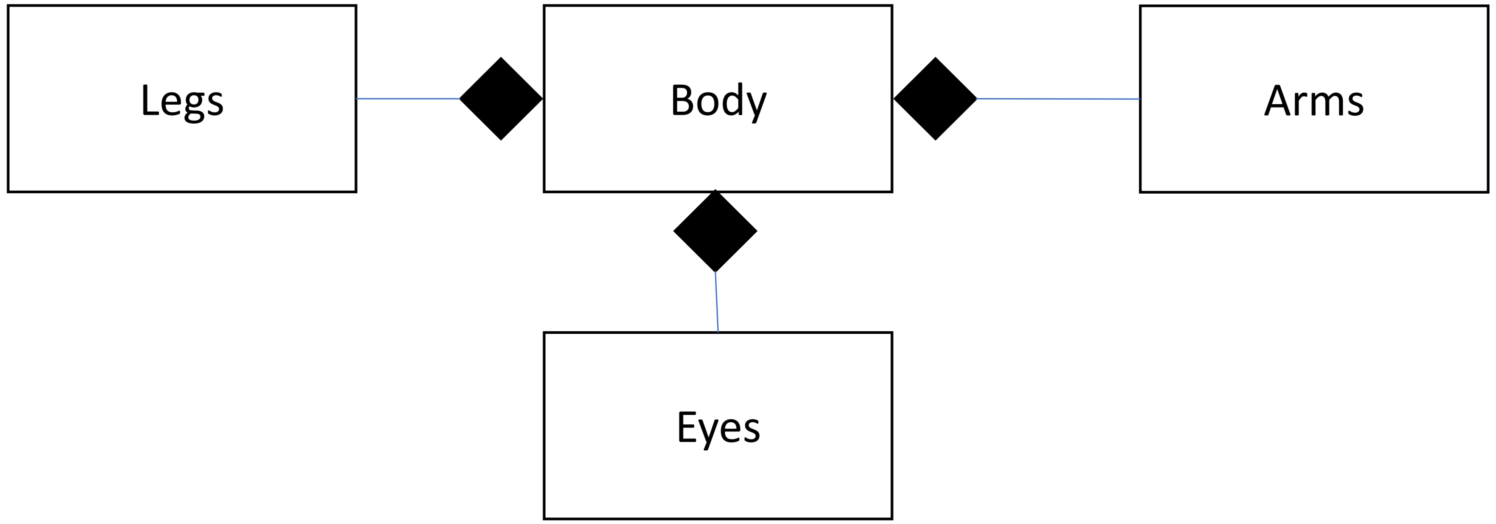
- Transitivity
- Anti-symmetry
- Propagation

Aggregation (part-of or part-whole relationship)

	Association	Aggregation	Inheritance
	has a relationship	A part of relationship	Is a relationship
	Between classes	Between parts and its assembly	Between parent and his child
	Multiplicity symbol used	Not used	Not used
	1-1, 1-*, *-*, *-1	fixed, variable, recursive	Single, multiple, multilevel, hierarchical, hybrid
	Not a kind of inheritance	Special kind of Association	Not kind of association or aggregation
	Denoted by ___	Denoted by ◇———	Denoted by ⟶
	A person works for a company	A paragraph consist of many sentences	Rectangle class can be derived from shape class

Composition (Strict whole relationship)

- Specialized form of aggregation
- Child objects does not have their lifecycle, if parent object deletes, all the child object will also be deleted.
- Composed object cant live independently.



Relationships Comparison

Association (has a)

- Definition: Association represents a basic relationship between two classes, showing that one class is related to another class. It is the weakest form of relationship.

- Dependency: In an association, objects of one class are not dependent on objects of another class. Changes in one class do not directly affect the other.

- Lifespan: The lifespan of both classes in the association is independent. They can exist independently of each other.

- Ownership: No ownership is implied in an association. The objects in each class are not owned by the other class.

- Example: Consider a Library and Books. A library contains books, but a book can exist outside the library.

Aggregation (part-of or whole-part)

- Definition: Aggregation represents a "whole-part" relationship between a class and its parts. It's a more specialized form of association.
- Dependency: In aggregation, the whole (aggregator) and its parts (aggregate) are somewhat dependent, but not fully. If the whole is destroyed, the parts can still exist.
- Lifespan: The lifespan of the whole is independent of the lifespan of the parts, but the whole manages the parts.
- Ownership: Aggregation implies that the whole class (aggregator) has ownership of the parts (aggregate). The parts can belong to, or be shared among, multiple wholes.
- Example: Think of a University and Departments again. A department is part of the university, but it can exist independently. The university owns and manages the departments. Each department might contain faculty members, and the lifespan of faculty members isn't tied to the university.

Composition (strict whole-part)

- Definition: Composition is a stronger form of aggregation, representing a strict "whole-part" relationship where the parts are exclusively owned by the whole.

- Dependency: In composition, the whole and its parts are highly dependent. If the whole is destroyed, the parts cease to exist.

- Lifespan: The lifespan of the whole and the parts is interdependent. When the whole is created, it also creates the parts. When the whole is destroyed, the parts are destroyed.

- Ownership: Composition implies that the whole class has exclusive ownership of its parts. The parts cannot belong to other wholes.

- Example: Take a Computer and its Motherboard. A computer contains a motherboard, and the motherboard is an integral part of the computer. If the computer is dismantled or scrapped, the motherboard is also unusable. The motherboard is exclusively owned by the computer.

Implementation

In C++, association, aggregation, and composition are ways to represent relationships between classes.

1. Association:

1. Represents a bi-directional relationship between two classes.
2. Objects of one class are related to objects of another class, but they are independent and can exist without each other.
3. Usually implemented using member variables or methods that refer to objects of another class.

Implementation

2. Aggregation:

1. Represents a "has-a" relationship where one class contains another class, but they can exist independently.
2. The child class (part) can be shared among multiple parent classes (whole).
3. Typically implemented using pointers or references.

3. Composition:

1. Represents a strong ownership relationship where one class contains another class, and the contained class cannot exist independently.
2. The child class (part) is created and destroyed as part of the parent class (whole).
3. Implemented by directly including objects of another class as member variables.

Good Luck!