UNIVERSITY OF SCIENCE
HO CHI MINH CITY

Inheritance

Nguyen Van Vu
nvu@fit.hcmus.edu.vn

Topics

- Inheritance
- Inheritance in C++
- Overriding

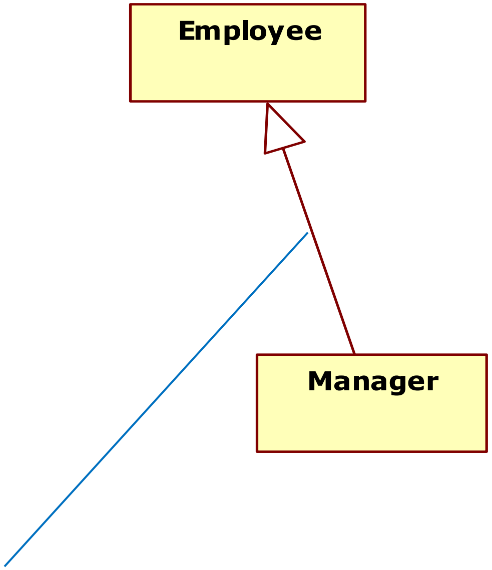
Inheritance

- A mechanism for incorporating structure and behavior of **general** elements to more **specific** elements
  - E.g., a manager is an employee of a company
    - Manager is more specific than employee, thus, manager inherits structure (attributes) and behavior (operations) of employee
- One element can inherit one or multiple more general elements

Inheritance

- Inheritance is the implementation of the **generalization relationship**

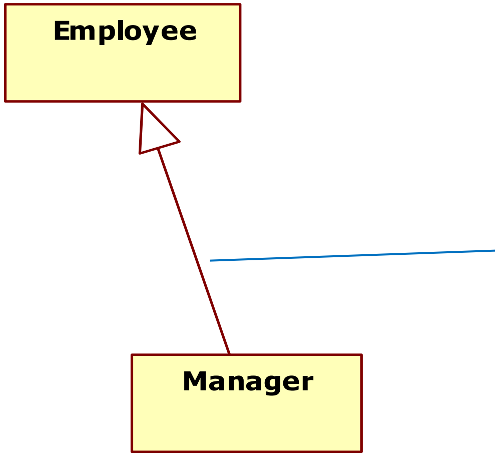
- Main purpose of inheritance
  - Reusability
    - Reuse attributes
    - Reuse operations



Generalization relationship
(also called “is a” relationship)

Inheritance

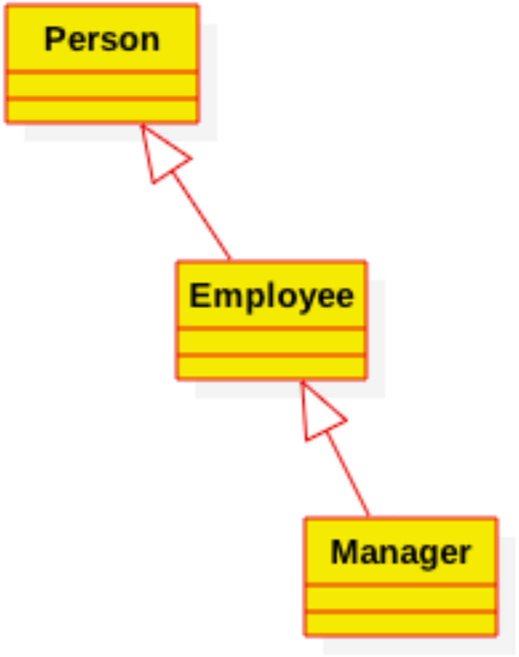
- A specific class inherits one or more general classes
- Child class or subclass is more specific than parent or superclass
- Subclass is consistent with superclass and has more information
  - E.g., any manager is an employee



Generalization relationship
(also called “is a” relationship)

Inheritance

- Superclass = base class = parent
- Subclass = derived class = child class
- Ancestor
- Descendant



- Generalization is a transitive relationship
  - Person is a superclass of Manager

Inheritance

- Superclass describes common structure and behavior of all of its descendants

- Subclass incorporates structure and behavior combined from its ancestors
  - An instance of a subclass may be used where its ancestors are allowed
  - The opposite is not possible → anti-symmetric relationship

Inheritance in C++

- Declaration

  class **<derivedclass> :** <*inheritance type*> **<baseclass>**

- Inheritance types: public, private, protected

```
class Employee : public Person
{
private:
        // New attributes of A.
public:
        // New methods of A.
};
```

Inheritance in C++

```
class Person {
private:
        string m_Name;
        int     m_Age;
        string m_ID;
public:
        Person(string name, int age)
        string getName();
}
class Employee : public Person {
private:
        float m_Salary;
public:
        float getSalary();
}
```

```
main() {
        Person p1 (”Minh”, 19).
        Employee e(“Anh”, 22)

        cout << p1.getName();
         cout << e.getName();

        p1.getSalary() ???

}
```

Anti-symmetric relationship

```
main() {
        Person p1 (”Minh”, 19).
        Employee e(“Anh”, 22)

        cout << p1.getName();
        cout << s.getName();

        Person p2 = e;
        Person p3 = Employee(”Nhan”, 25);

        Employee e2 = Person(“Hung”, 30); //wrong
        Employee e3 = p1; //wrong

}
```

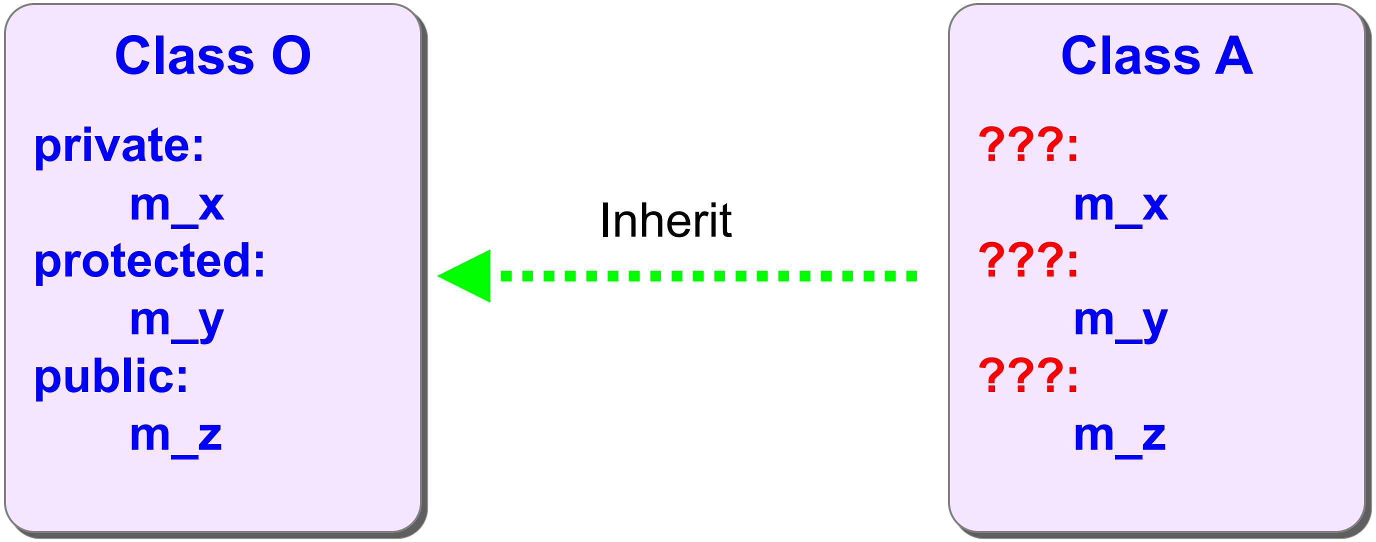
- An instance of a subclass may be used where its ancestors are allowed
- The opposite is not possible → anti-symmetric relationship

Practice

- In our Student and Course case, suppose we have Teacher. Teacher and Student are a person.
- Write class Person and change class Student using Inheritance

Access control in inheritance

- Class A inherits from class O:
  - A inherits all attributes and methods from O.
  - Do scopes changed during inheritance?



Class O

private:
    m_x
protected:
    m_y
public:
    m_z

Inherit

Class A

???:
    m_x
???:
    m_y
???:
    m_z

➔ **Decided by inheritance type!!**

Access control in inheritance

- Access control in inheritance:

Scope	public inheritance	protected inheritance	private inheritance
public	public	protected	private
protected	protected	protected	private
private	***inaccessible***	***inaccessible***	***inaccessible***

Method everriding

- A method from superclass can be written or redefined
- The overriding method in subclass must has **the same signature** as the superclass' method

```
class Employee : public Person {
public:
        float getSalary();
}

class Manager : public Employee {
public:
        float getSalary(); //overriding method
}
```

Overloading and Overriding

- What is the difference between them?
- Overloading – member methods of the same name but different parameters
- Overriding – derived methods of the same name and parameters with the parent classes (same signature)

Superclass’ default constructor is called by default

```
class Person {
private:
        string m_Name;
        int    m_Age;
        string m_ID;
public:
        Person(string name, int age)
        string getName();
}

class Employee : public Person {
private:
        float m_Salary;
public:
        float getSalary();
}
```

```
main() {
        Person p1 (”Minh”, 19);
        Employee e(“Anh”, 22);

        cout << p1.getName();
        cout << e.getName();

        Employee e2;
}
```

How do you call a constructor of the superclass?

```
class Person {
private:
        string m_Name;
        int    m_Age;
        string m_ID;
public:
        Person(string name, int age)
        string getName();
}

class Employee : public Person {
private:
        float m_Salary;
public:
        Employee(string name, int age, float salary); //???
        float getSalary();
}
```

Here you want to set name, age, and salary.

How do you call a constructor of the superclass?

```
class Person {
private:
        string m_Name;
        int     m_Age;
        string m_ID;
public:
        Person(string name, int age)
        string getName();
}

class Employee : public Person {
private:
        float m_Salary;
public:
        Employee(string name, int age, float salary) : Person(name, age) {
            m_Salary = salary;
        }
        float getSalary();
}
```

How to call a method of the superclass?

```
class Person {
private:
        string m_Name;
        int    m_Age;
        string m_ID;
public:
        Person(string name, int age)
        string getName();
}

class Employee : public Person {
private:
        float m_Salary;
public:
        void printSalary () {
                cout << Person::getName() << m_Salary < endl;
        }
}
```

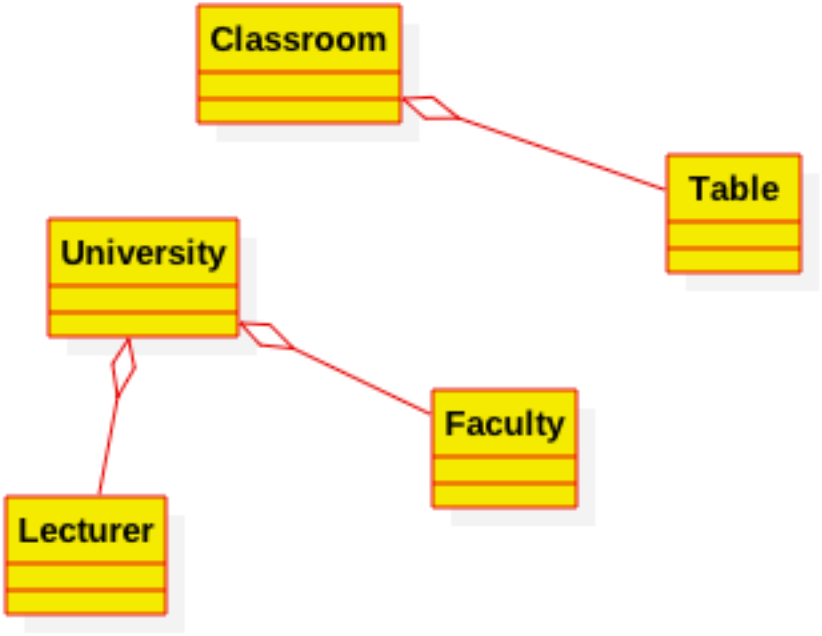
Is-a and Has-a relationships

- Generalization is sometimes called "is-a" relationship
- A derived class is a special case of the superclass
  - An employee **is a** person
  - A cat **is an** animal

- Aggregation is called "**has-a**" relationship
  - A classroom has a board
  - A classroom has tables
  - A page is a part of a book
  - A book contains pages

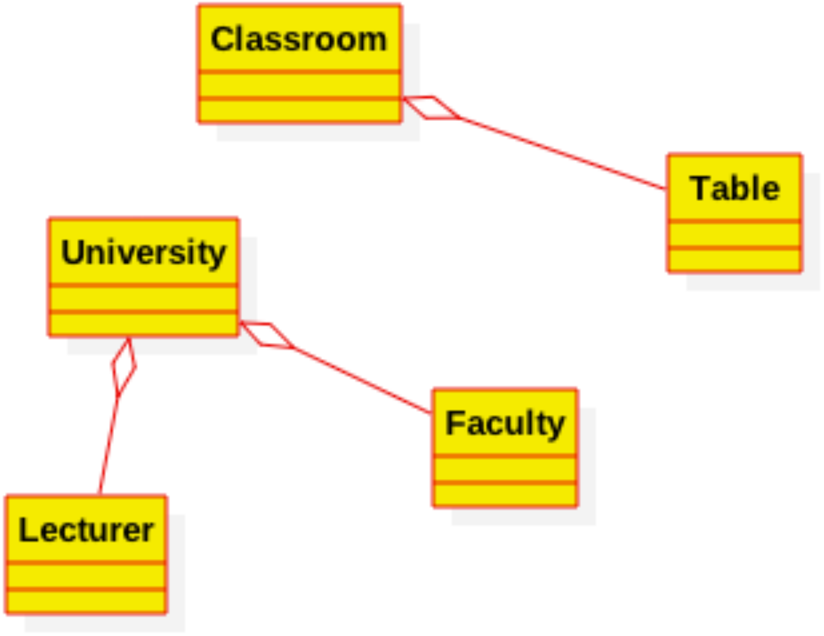
Aggregation in UML

- Aggregation is a very common relationship
- How do you implement this relationship in C++?



Aggregation in UML

- How do you implement this relationship in C++?
  - Two approaches
    - Use a member variable to contain a list of objects
    - Use a member variable to store another object



Initialization order of derived objects

- 1. Base class' constructor is called first

  ➔ **Initialize inherted members**
- 2. Derived class' constructor is called after

  ➔ **Initialize new members**
- 3. Derived class can **decide** how to initialize its core

  ➔ **Identify base class constructor to call**

  ➔ **Forget to identify: default construct is called**

Initialization order of derived objects

```
class Person {
private:
            string m_Name;
            int       m_Age;
            string m_ID;
public:
            Person(string name, int age)
            string getName();
}

class Employee : public Person {
private:
            float m_Salary;
public:
            Employee(string name, int age, float salary) : Person(name, age) {
            //…
            }
            float getSalary();
}
```

Initialization order example

```
class A {
 public:
        A( int x ) {   }
};

class B: public A {
public:
        B( ) {   }
        B( int x, int y ): A( x ) { }
};

class C: public B {
public:
        C( ) {   }
        C( int z ) {   }
        C( int x, int y, int z ): B( x, y ) {   }
};
```

Point out initialization order of the followings:

```
a) void main() {   C   obj( 1, 2, 3 );   }
b) void main() {   C   obj( 4 );         }
c) void main() {   C   obj;              }
```

Destruction order of derived object

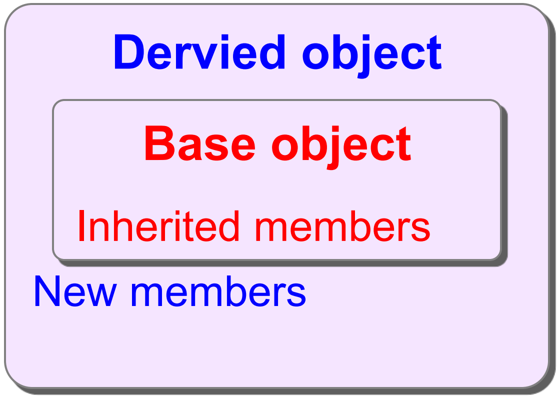
- Initialization order in reverse
  - 1. Derived class' destructor is called first

    ➔ **Dispose object's skin**
  - 2. Base class' destructor is called after

    ➔ **Dispose object's core**

  - Class has only 1 destructor

    ➔ **Identifying the destructor is not necessary**



Dispose from skin to core

Practice

Draw inheritance tree for the following classes:

(create base classes needed for reusability)

- Square.
- Circle.
- Ellipse.
- Rectangle.
- Diamond.
- Parallelogram.
- Isosceles trapezoid.
- Right trapezoid.
- Right triangle.
- Isosceles triangle.
- Right isosceles triangle.
- Equilateral triangle.