



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

CSCI251

Advanced Programming

Week 7: Inheritance and Polymorphism



Object Oriented Programming in C++

Outline

Inheritance

1. What is it?
2. Multilevel Inheritance
3. Multiple Inheritance
4. Protected
5. Why inheritance?

Polymorphism

6. What is Polymorphism?
7. Virtual Functions

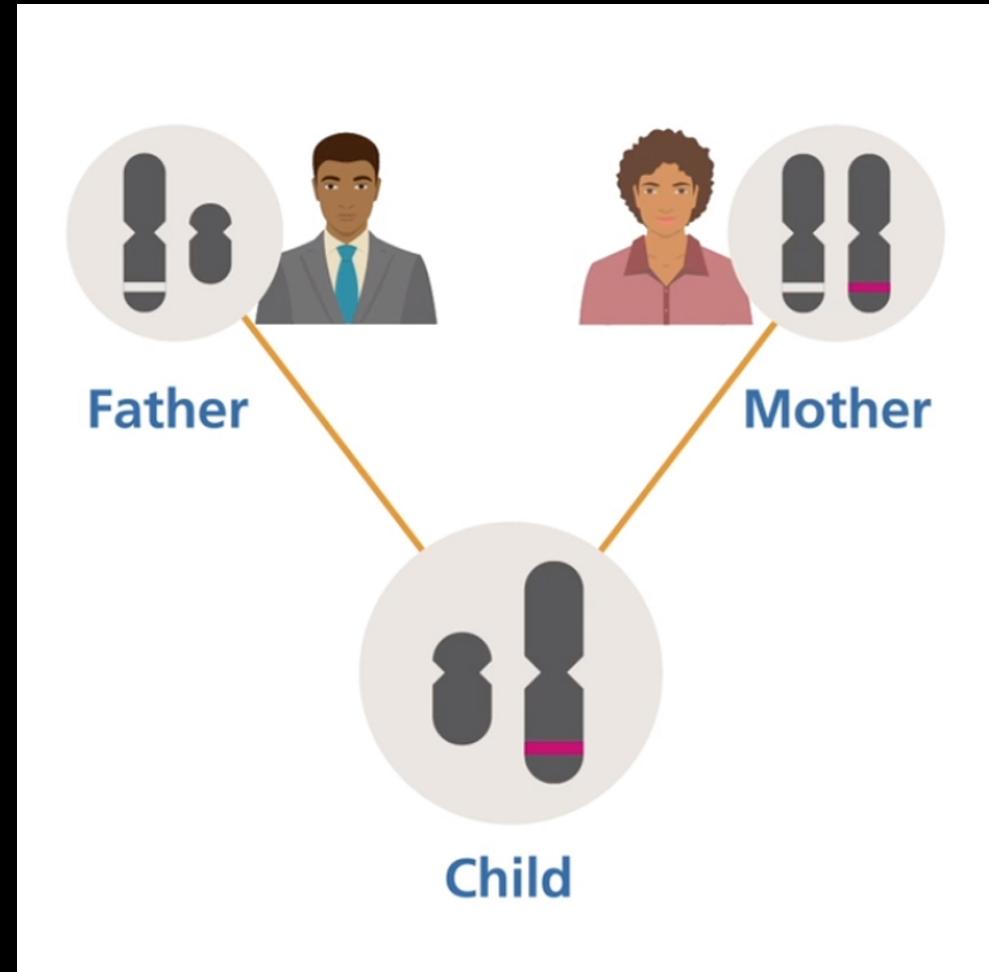
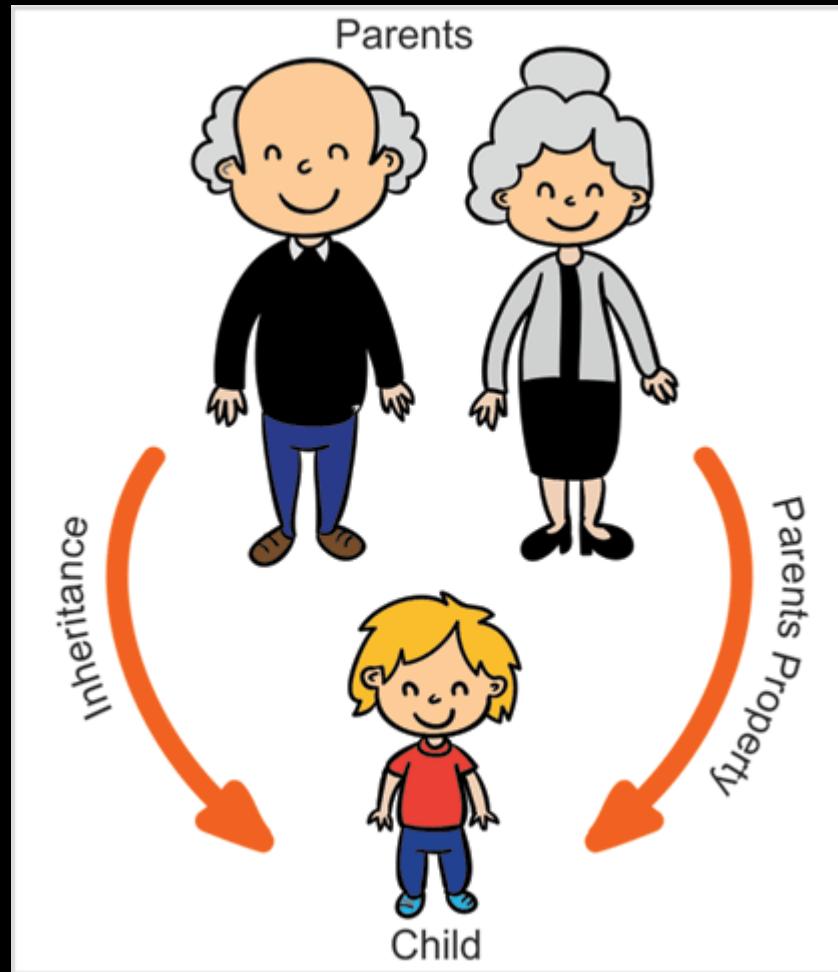
UML Language

8. What is UML?
9. Class/Object Relations



1. Inheritance

What is it?



Source: Code Stall

What is Inheritance?

Concept: Inheritance allows us to **define** a child class that **inherits** all the methods and properties from a parent class.

We have two categories here:

Child (derived class): the class that inherits from another class

Parent (base class) : the class being inherited from

What is Inheritance?

Syntax: class Child_Class: public Parent_Class{...};

```
class White_Hacker: public Hacker {};
```

Inheritance

class Hacker:

Attributes:

**name
level**

Methods:
scan()
attack()

```
class White_Hacker: public Hacker
```

class White_Hacker:

Attributes:

**name
Level**

Methods:
scan()
attack()
defend()

Parent Class

Child Class

Code

```
#include <iostream>
using namespace std;
class Hacker {      // The class Hacker
public:             // Access specifier
    string name;   // Attribute (string variable)
    int level;     // Attribute (int variable)
    void scan(){   // Create function/method inside class
        cout << "I am scanning the target";
    };
};
class White_Hacker: public Hacker{
public:
    void defend(){
        cout << "----for defending!----";
    }
};
int main(){
    White_Hacker w_Hacker; // an Object of White Hacker
    w_Hacker.name = "John";
    w_Hacker.level = 7;
    cout << "Hacker "<< w_Hacker.name << ", level: "<<w_Hacker.level<<"\n";
    w_Hacker.scan(); // Call the method
    w_Hacker.defend();
    return 0;
}
```

Practice 1: Inheritance

- Do in Code:Block
- Create a class Gray_Hacker inherited from class Hacker with one method (use public: as access specifier)
 - void purpose()
 - print to screen “---for YOUR WORDs-----”
- In main() function:
 - Create an object gray_Hacker of class Gray_Hacker
 - Call the method scan()
 - Call the method purpose()

Multilevel Inheritance

Multilevel Inheritance?

Concept: A class can also be derived from one class, which is already derived from another class.

`Child_Hacker` is derived from class
`White_Hacker`, which is derived from
class `Hacker`

Code

```
#include <iostream>
using namespace std;

class Hacker {      // The class Hacker
public:           // Access specifier
    string name;   // Attribute (string variable)
    int level;     // Attribute (int variable)
    void scan(){   // Create function/method inside class
        cout << "I am scanning the target";
    };
};

class White_Hacker: public Hacker{
public:
    void defend(){
        cout << "----for defending!----\n";
    };
};

class Child_Hacker: public White_Hacker {
public:
    void whoiam(){
        cout << "I am a child of a white Hacker";
    };
};

int main(){
    Child_Hacker ch_Hacker; // Create an Object of Child Hacker
    ch_Hacker.scan();
    ch_Hacker.defend();
    ch_Hacker.whoiam();
    return 0;
}
```

Practice 2: Multilevel Inheritance

- Do in Code:Block
- Create a class `Child_Hacker` inherited from class `Gray_Hacker` with one method (use `public:` as access specifier)
 - `void whoiam()`
 - print to screen “I am a child of a Gray Hacker”
- In `main()` function:
 - Create an object `ch_Haker` of class `Child_Hacker`
 - Call the method `scan()`
 - Call the method `purpose()`
 - Call the method `whoiam()`

Multiple Inheritance

Multiple Inheritance?

Concept: A class can also be derived from more than one base class, using a comma-separated list.

Child_Hacker is derived from class Ethical, and White_Hacker, which is derived from class Hacker

Code



```
#include <iostream>
using namespace std;
class Hacker { // The class Hacker
public: // Access specifier
    string name; // Attribute (string variable)
    int level; // Attribute (int variable)
    void scan(){ // Create function/method inside class
        cout << "I am scanning the target";
    }
};
class White_Hacker: public Hacker{
public:
    void defend(){
        cout << "----for defending!----\n";
    }
};
// Another class
class Blue {
public:
    void blue(){
        cout << "I am also doing in a Blue Team";
    }
};
// Multiple Class
class Child_Hacker: public White_Hacker, public Blue {
public:
    void whoiam(){
        cout << "I am a child of a white Hacker\n";
    }
};
int main(){
    Child_Hacker ch_Hacker; // Create an Object of Child Hacker
    ch_Hacker.scan();
    ch_Hacker.defend();
    ch_Hacker.whoiam();
    ch_Hacker.blue();
    return 0;
}
```

Practice 3: Multiple Inheritance

- Do in Code:Block
- Create a class Red with one method red() print out is “I am also doing in a Red Team”
- Create a class Child_Hacker inherited from class Gray_Hacker and Red class
- In main() function:
 - Create an object ch_Haker of class Child_Hacker
 - Call the method scan()
 - Call the method purpose()
 - Call the method whoiam()
 - Call the method red()

Access Specifier: **protected**

Access Specifier - protected

Recall: To modify the access feature of attributes and methods, we have 3 different types of access specifiers:

- Public can be directly accessed from outside the object.
- Private can only be directly accessed by internal methods.
- Protected can be directly accessed by objects of subclasses (inheriting class), but not by arbitrary external objects.

Access Specifier - protected

- Protected is similar to private, but it can also be accessed in the derived class (subclass).

Code

```
#include <iostream>
using namespace std;

class Hacker {      // The class Hacker
protected:          // Access specifier
    int level;     // Attribute (int variable)
};
class White_Hacker: public Hacker{
public:
    int upgrade_Level;
    void setLevel(int l){
        level=l;
    }
    int getLevel(){
        return level;
    }
};
int main(){
    White_Hacker w_Hacker; // Create an Object of White Hacker
    w_Hacker.setLevel(7);
    w_Hacker.upgrade_Level = 2;
    // Print to the screen the values
    cout << "White hacker has level: " << w_Hacker.getLevel() << "\n";
    cout << "now increase " << w_Hacker.upgrade_Level << " levels\n";
    cout << "The level now is " << w_Hacker.getLevel() + w_Hacker.upgrade_Level;
    return 0;
}
```

Why we use Inheritance?

- Code reusability:
 - reuse attributes and methods of an existing class when you create a new class.

Summary: Access to data and member functions

Access	Public	Protected	Private
Same class			
Derived classes			
Outside classes			

Summary: Access to data and member functions

Access	Public	Protected	Private
Same class	YES	YES	YES
Derived classes	YES	YES	NO
Outside classes	YES	NO	NO

Summary: Inheritance and access to data and member functions

- **Public inheritance; deriving from public base class implies that**
 - **Public members of base class become public in derived class**
 - **Protected members of base class become protected members in derived class**
 - **Private members of base class are not accessible directly from derived class**

Summary: Inheritance and access to data and member functions

- . Private and protected inheritance**

Practice 4



Polymorphism - What is it?



“Many forms”

This is used when we have many classes that are related to each other by inheritance

This allows to perform single action in different ways

Example



```
class Animal  
+ animalSound()
```



```
class Pig  
+ animalSound()
```



```
class Dog  
+ animalSound()
```



```
class Cat  
+ animalSound()
```

Code

```
The animal makes a sound  
The pig says: wee wee  
The dog says: wow wow  
The cat says: meow meow
```

```
Process returned 0 (0x0)  
Press any key to continue.
```

Practice 5

```
#include <iostream>  
using namespace std;  
// Base class  
class Animal {  
public:  
    void animalSound(){cout << "The animal makes a sound \n" ;}  
};  
// Derived class  
class Pig : public Animal {  
public:  
    void animalSound() {cout << "The pig says: wee wee \n" ;}  
};  
class Dog : public Animal {  
public:  
    void animalSound() {cout << "The dog says: wow wow \n" ;}  
};  
class Cat : public Animal {  
public:  
    void animalSound() {cout << "The cat says: meow meow \n" ;}  
};  
int main() {  
    Animal myAnimal;  
    Pig myPig; Dog myDog; Cat myCat;  
    myAnimal.animalSound();  
    myPig.animalSound();  
    myDog.animalSound();  
    myCat.animalSound();  
}
```

Virtual function

Runtime polymorphism.
Ensure the correct function is called, regardless of the type of the pointer.

```
The pig says: wee wee  
The dog says: wow wow
```

```
#include <iostream>  
using namespace std;  
// Base class  
class Animal {  
public:  
    virtual void animalSound(){cout << "The animal makes a  
sound \n" ;}  
};  
// Derived class  
class Pig : public Animal {  
public:  
    void animalSound() {cout << "The pig says: wee wee \n" ;}  
};  
class Dog : public Animal {  
public:  
    void animalSound() {cout << "The dog says: wow wow \n" ;}  
};  
class Cat : public Animal {  
public:  
    void animalSound() {cout << "The cat says: meow meow \n" ;}  
};  
int main() {  
    Animal *ptr=new Pig();  
    ptr->animalSound();  
    ptr=new Dog();  
    ptr->animalSound();
```

3. UML



What is UML?

UML stands for “Unified Modelling Language”

- It is an industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design

Type of UML

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- State Diagram

Class Diagram

- Provide a conceptual model of the system in terms of classes and their relationships between them.
- Used for requirement capture, end-user interaction.
- Detailed class diagrams are used by developers.

Class Representation

- Each class is represented by a rectangle subdivided into three compartments
 - Name
 - Attributes
 - Operations
- Modifiers are used to indicate visibility of attributes and operations.
 - ‘+’ is used to denote Public visibility (everyone)
 - ‘#’ is used to denote Protected visibility (friends and derived)
 - ‘-’ is used to denote Private visibility (no one)
- By default, attributes are hidden and operations are visible.

Class Visualisation

class name

attributes

methods

Hacker

name: string

level: int

getName(): string

setName(string): void

getLevel(): int

Class Visualisation

Hacker
- name: string
- level: int
+ getName(): string
+ setName(string): void
+ getLevel(): int
+ setLevel(int): void

'+' public

'#' protected

'-' private

Class Visualisation

Hacker

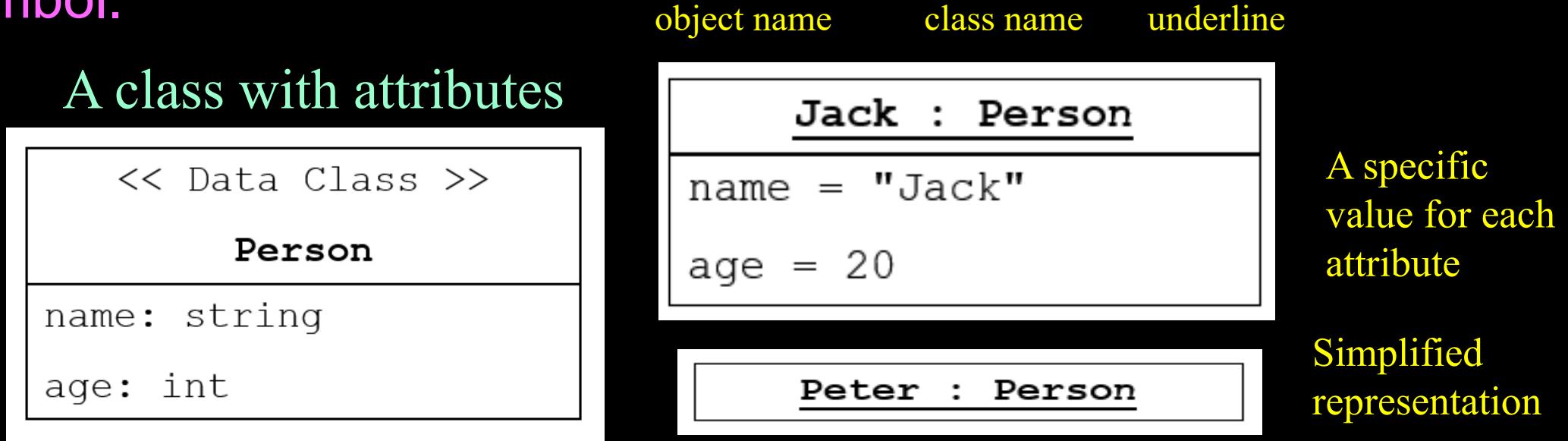
- **name: string**
- **level: int**

- + **getName(): string**
- + **setName(string): void**
- + **getLevel(): int**
- + **setLevel(int): void**

```
class Hacker {  
    private:  
        string name;  
        int level;  
    public:  
        string getName() {...}  
        void setName(String) {...}  
        int getLevel() {...}  
        void setLevel(int) {...}  
};
```

Object Diagram

- The building block is the object symbol, each based on a class symbol.



Class/Object Relations

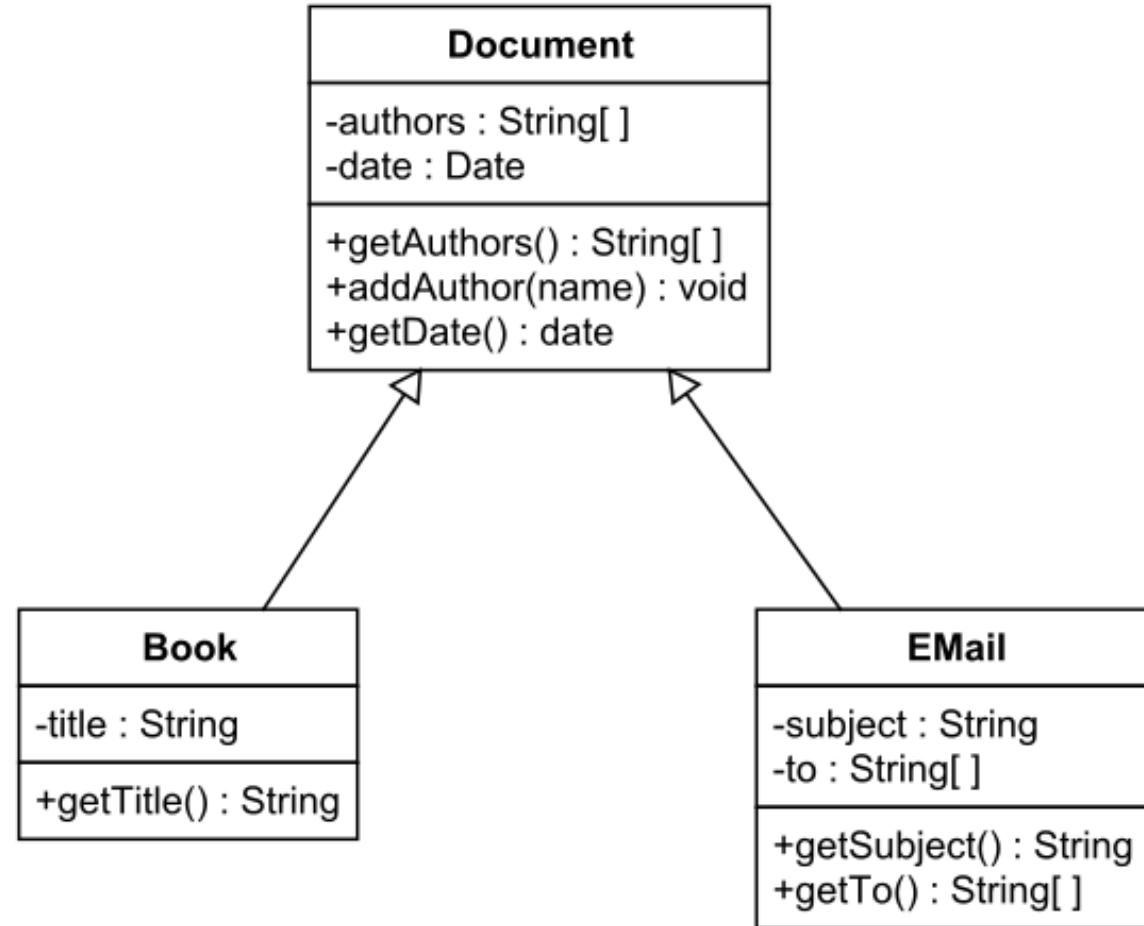
- Class symbols by themselves provides quite limited information about the system.
- A system will often have multiple classes related in various ways, and a model of the system should reflect such relationship.
- UML defines the following relationship types:
 - Inheritance
 - Dependency.
 - Association.
 - Aggregation.
 - Composition.

Class/Object Relations

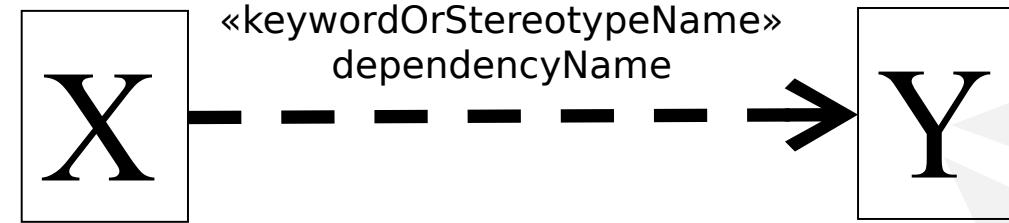
- The nature of the relationship can significantly impact how changes we make to one class, or to objects of that class, impact other classes or objects.
- In a certain sense those relations are in order of binding strength.

<http://www.uml-diagrams.org/>

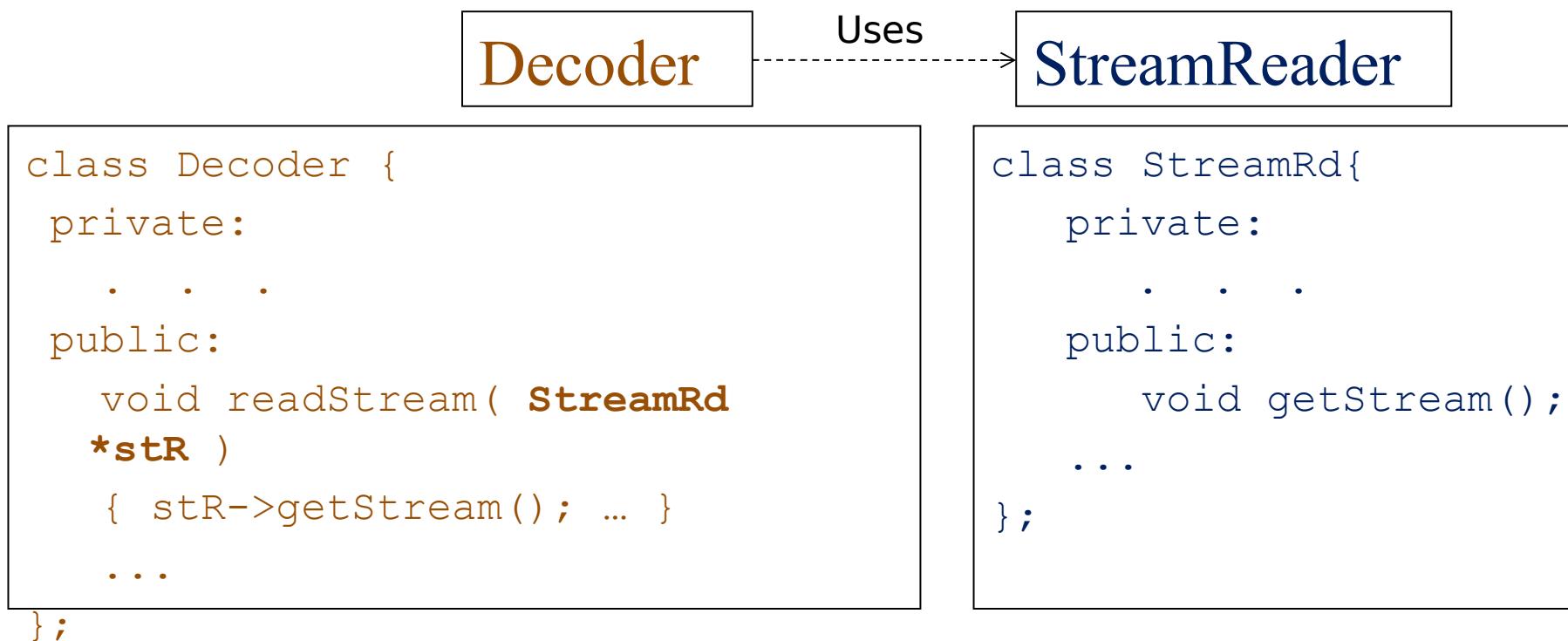
Inheritance



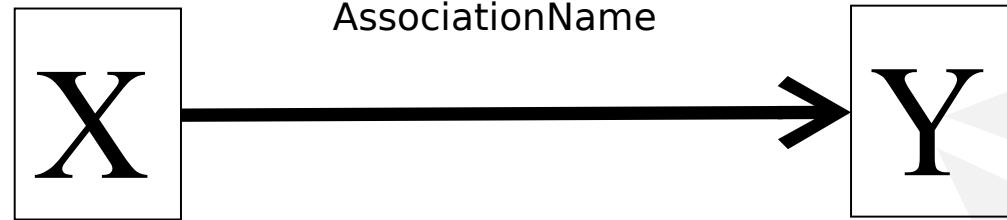
Dependency



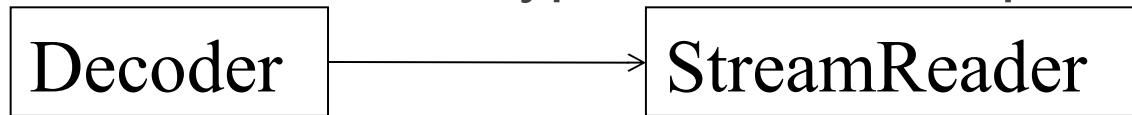
- Very general, one directional relationship indicating that one class uses another class, or depends on it in someway.
 - X uses (or depends on) Y but Y isn't influenced by X.



Association



- One class retains a relationship with another class.
 - Often a collection of object links.
 - The illustration is one directional but these are often bi-directional and then a line without arrows is used.
- Association can be described as a “has a” type of relationship.



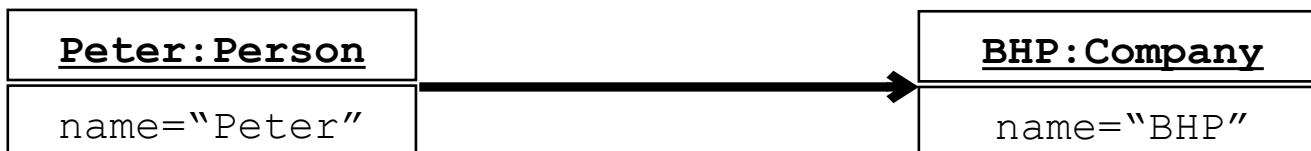
```
class Decoder {  
private:  
    StreamRd *stR;  
public:  
    Decoder();  
    Decoder( StreamRd *stream )  
    { stR = stream; ... }  
    ...  
};
```

```
class StreamRd{  
    . . .  
};
```

- In the association implementation we have an object of one type *being* in another class, as opposed to the object being based in a function of the class as seen in the dependency.
- Association, aggregation, and composition all typically have this property; the distinction between them is based around the type of connection.
 - With a dependency it's more like using the functionality of another class instance.
 - cout, an instance of ostream, is an example of dependency.

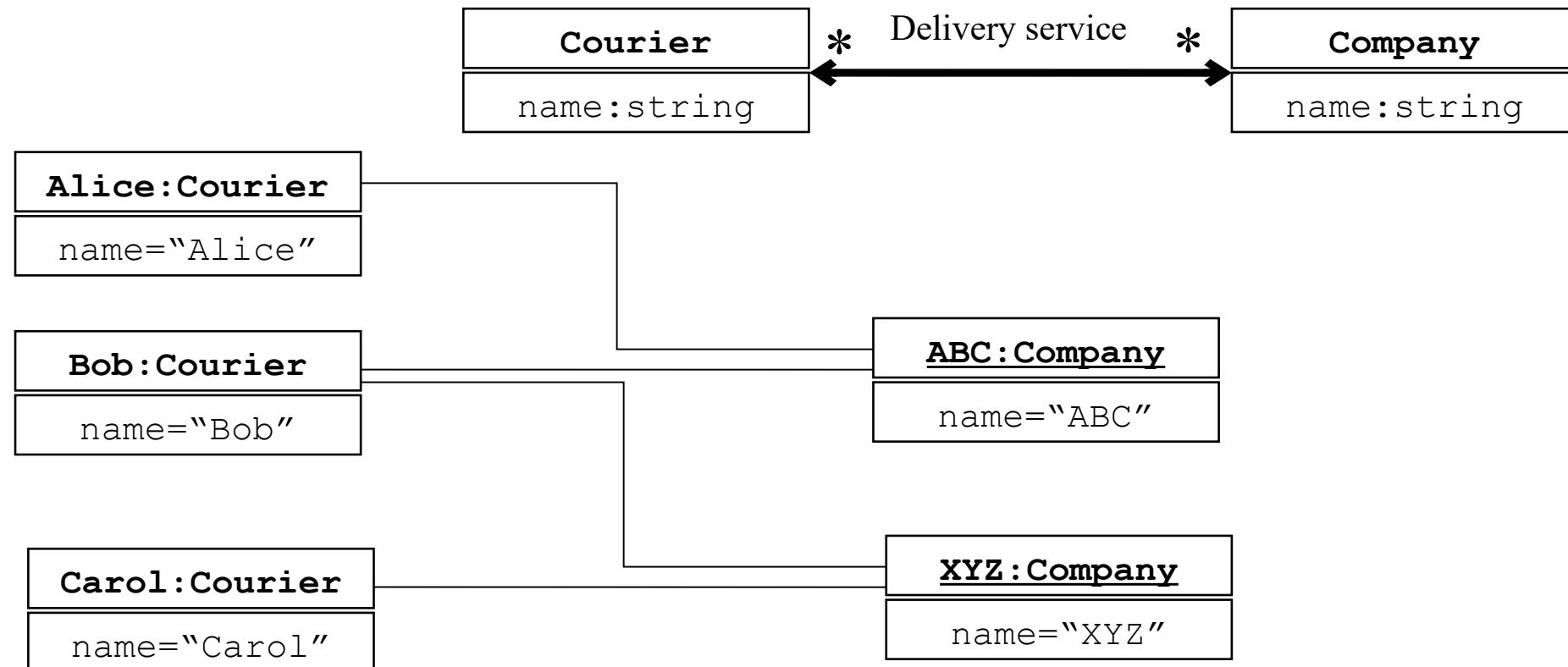
Object links and Class associations

- An association in a class diagram is represented by links between objects in an object diagram, showing that the objects are related/communicate with each other.
 - You can link objects together so long as there is a relationship between their classes.
- The links reflect only a static view of the interaction between objects.

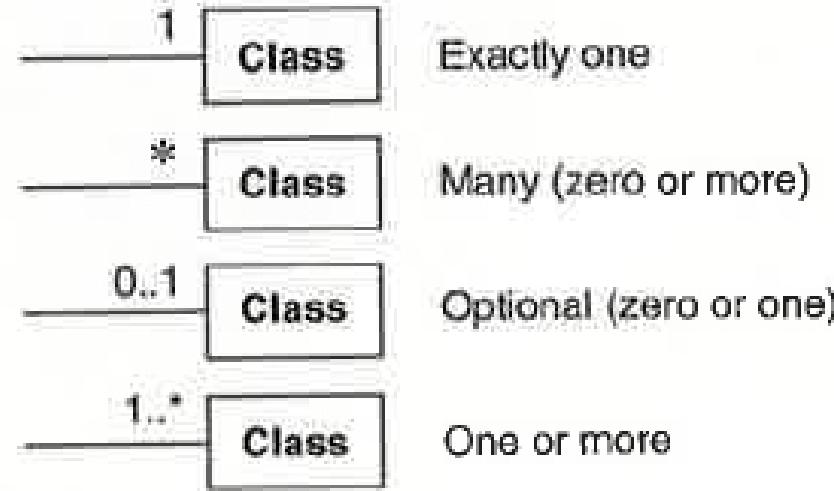


Linking multiple objects

- Sometimes we want to link multiple objects from one class to an object of another.
 - We illustrate this in a UML diagram using the multiplicity, represented by a *.



- From Blaha & Rumbaugh: **Multiplicity of Associations:**



- For example:

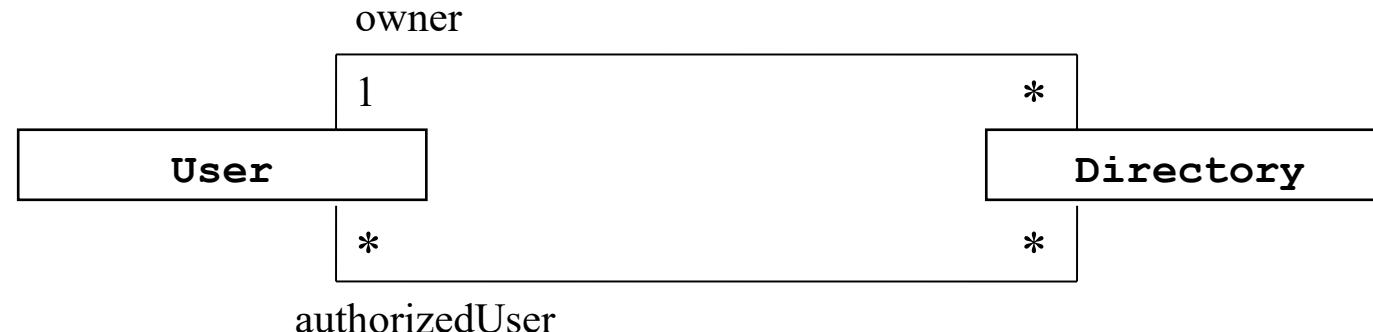
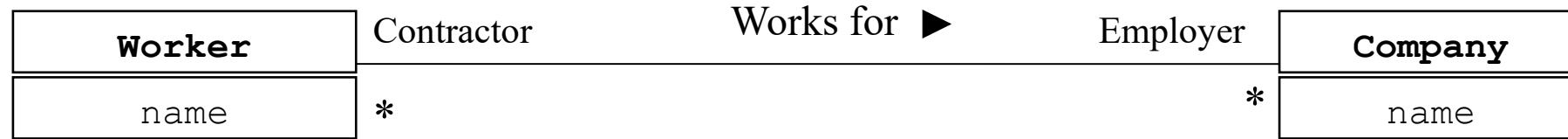


- Actually some countries have more than one capital.
- Countries have many cities, and we can use this to illustrate a range of values allowed for a multiplicity, in general “a..b”.



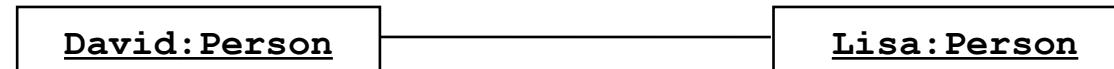
Labelling and multiple associations

- Sometimes the association name may not clearly describe the relationship between classes
- We can label the association roles to make things clearer.

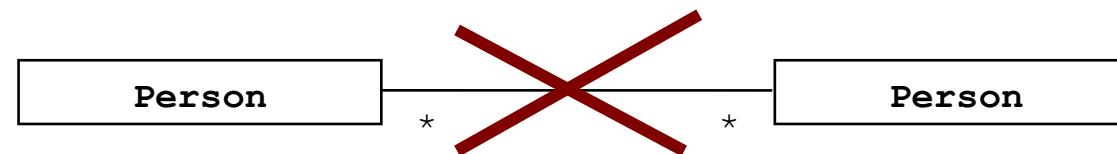


Self association

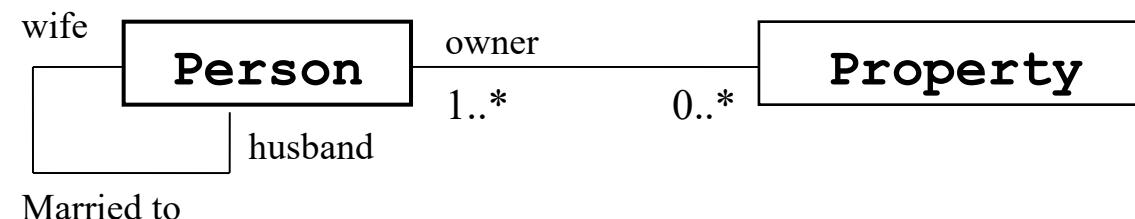
- Linked objects can be from the same class.



- However, the class diagram below is confusing.



- So, if a class can be associated to itself, only one class symbol should be shown.

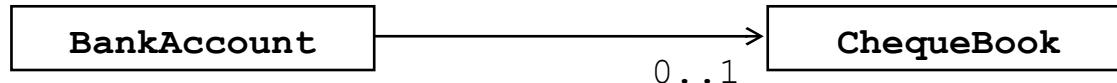


Some Implementation Examples

- Optional one-way Association:

Example: A Cheque Book may be issued for use with a Bank Account, but some account holders may not be interested.

The ChequeBook “doesn’t exist” independent of the BankAccount.

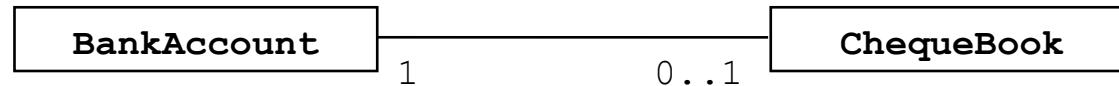


```
class BankAccount {  
    private:  
        ChequeBook *chBook;  
    public:  
        BankAccount (CheckBook *cP);  
        ...  
};
```

```
class ChequeBook {  
    ...  
};
```

- You need to keep track of whether a ChequeBook exists or not, if it doesn't the chBook would effectively be nullptr.

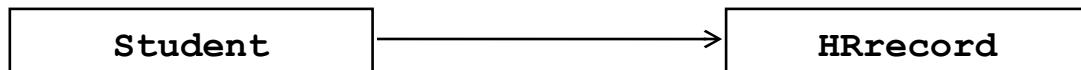
- Bi-directional Association (one-to-optional):
 - We could have a Cheque Book for the Bank Account, as before, but...
 - ... the Cheque Book has to be linked to a Bank Account.



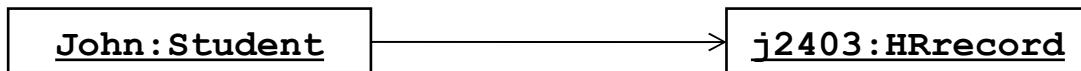
```
class BankAccount {  
    private:  
        ChequeBook *chBook;  
    . . .  
};  
  
class ChequeBook {  
    private:  
        BankAccount *account;  
    . . .  
};
```

The code illustrates the implementation of the associations shown in the UML diagram. The **BankAccount** class contains a private pointer **chBook** of type **ChequeBook**. The **ChequeBook** class contains a private pointer **account** of type **BankAccount**. A blue dotted oval encloses the declaration of **chBook** in the **BankAccount** class, and a blue dotted arrow points from this oval to the declaration of **account** in the **ChequeBook** class, visually connecting the two associations.

- One way link between objects



Class
diagram

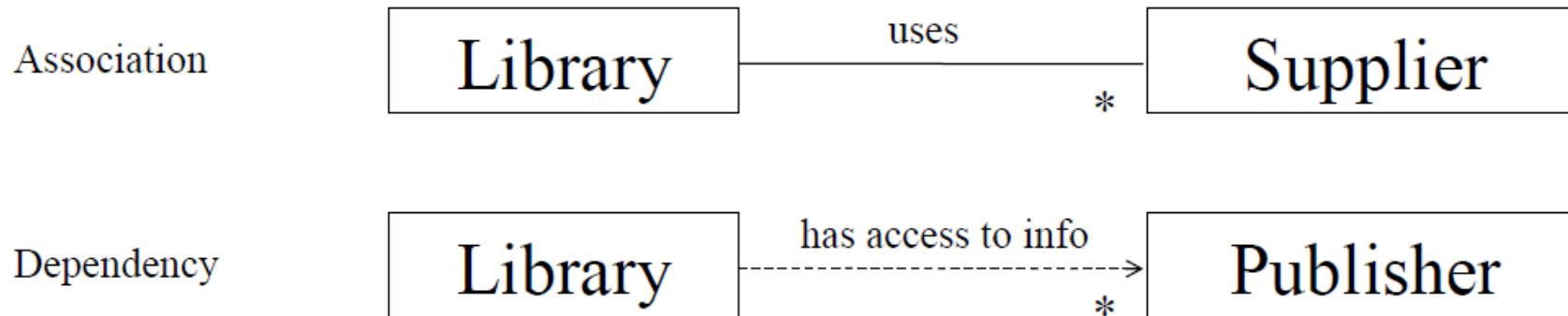


Object
diagram

```
class HRrecord{  
    . . .  
};  
  
class Student{  
    HRrecord *record;  
    . . .  
};  
.  
.  
.  
HRrecord *j2403 = new HRrecord();  
Student *John = new Student();  
  
John->addRecord( j2403 );
```

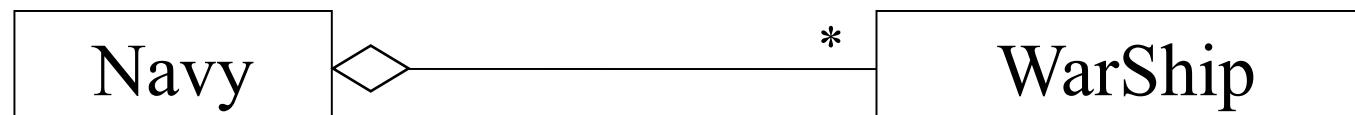
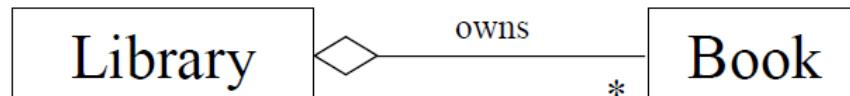
Association, Aggregation and Composition

- Aggregation and composition are subsets of association, and reflect "owns" relationship
- Aggregation implies that the two associated classes can exist independently.
- Composition implies that the two associated classes cannot exist independently.
- The association below does not indicate ownership.



Aggregation

- Examples: Class and Student, Parent and Child, Library and Book, Navy and WarShip



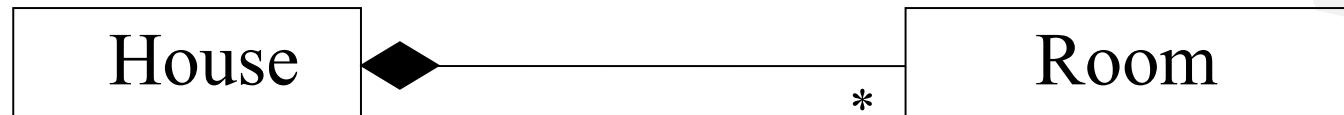
```
class Navy {  
private:  
    int currentlyInService;  
    WarShip *ships; // array of ships  
public:  
    void addnewShip( WarShip *nShip );  
    void decommissionShip( WarShip *nShip );  
    ...  
};
```

```
class WarShip{  
    . . .  
};
```

The implementation of aggregation is similar to association, it's a special case.

Composition or composite aggregation

- Composition is the strongest version of Association
 - the owned object of a class cannot live on its own. E.g., House owns rooms

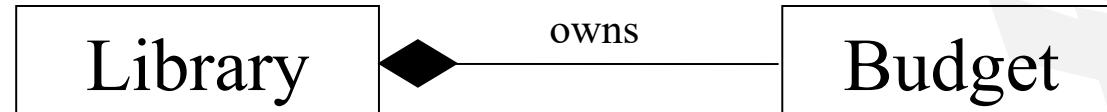


```
class House {  
private:  
    Room *rooms;  
public:  
    Cube () {  
        rooms = new Room [2];    }  
    ~Cube () {  
  
        delete [] rooms;  
    }  
    ...  
};
```

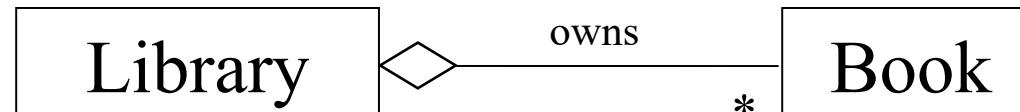
```
class Room{  
    . . .  
};
```

The relations to date ...

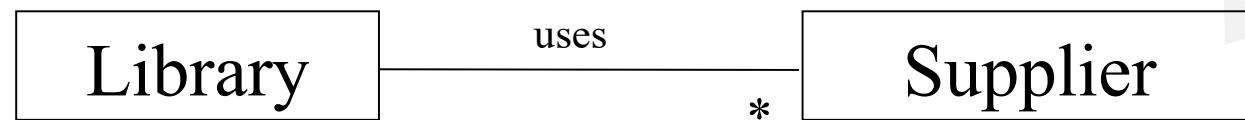
Composition



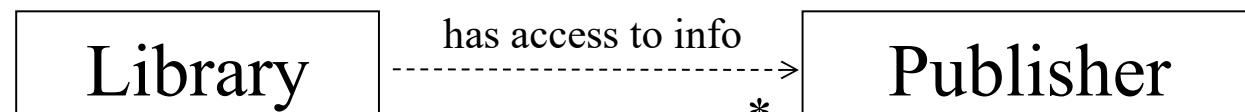
Aggregation



Association



Dependency

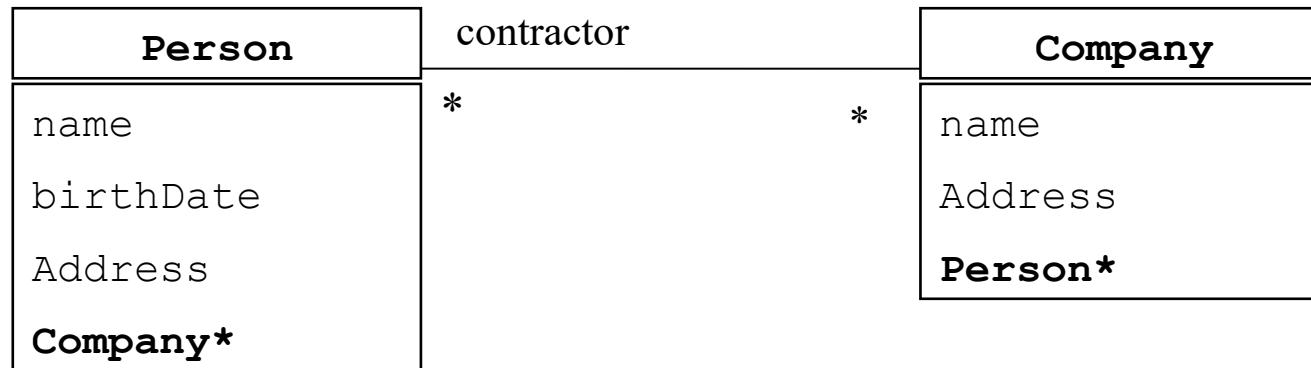


Inheritance



Association classes

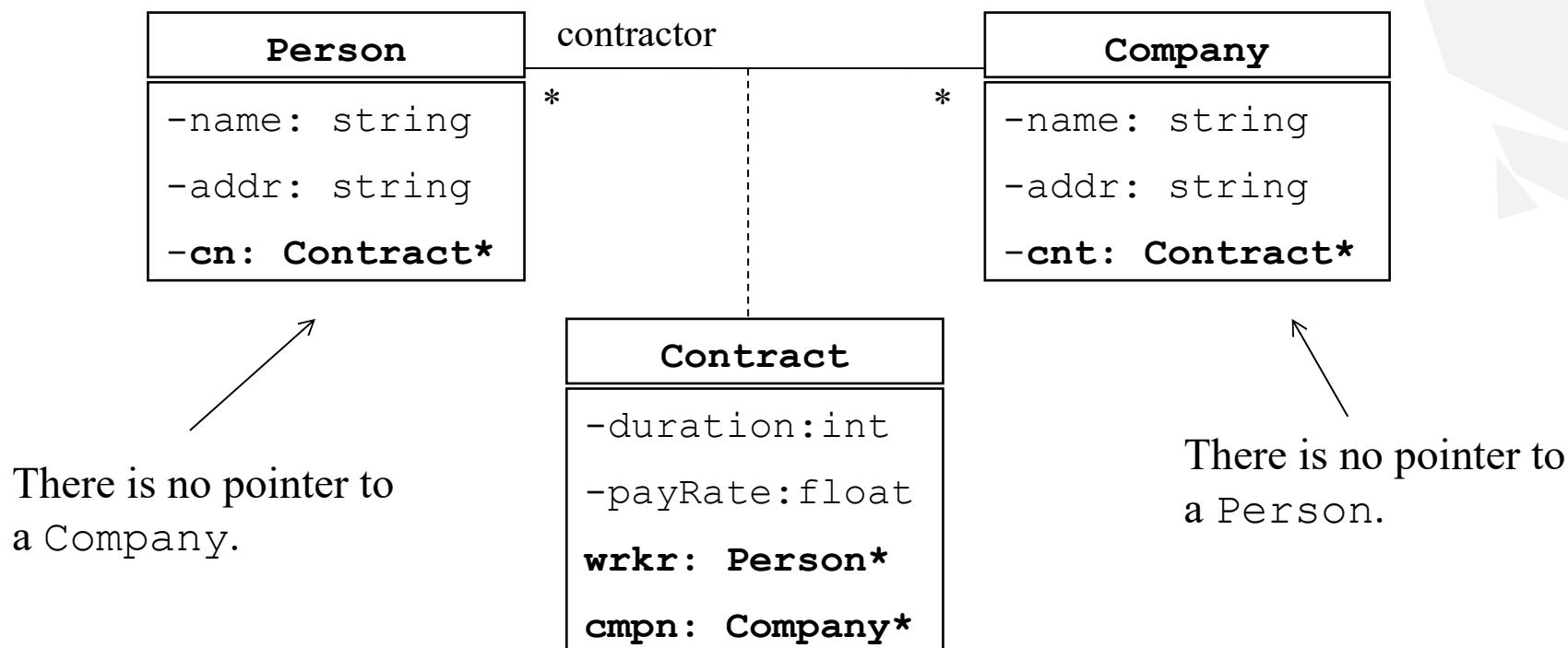
- Simple association can be implemented through pointers to objects as class data members.



- But if the association is implemented through a formal contract, where should the contract attributes be placed?
 - In the class Person?
 - In the class Company?
- How many contracts can a company have?
- Technically possible, but likely to be confusing and inefficient.

Association classes

- Complex associations can be modelled through classes.
- The notation for an association class is a class attached to an association by a dashed line.



```
#include <iostream>
#include <string>
using namespace std;

class Person;
class Contract;
class Company
{
    string name;
    string addresss;
Contract *contract; //association via Contract class public:
    Company(string Name="") : name(Name), contract(NULL) { }
    string getName() const { return(name); }
    void setContract(Contract *cn) { contract = cn; }
    Contract *getContract() const { return( contract); }
};
```



```
class Person
{
    string name;
    Contract *contract; //association via Contract class
public:
    Person(string Name="") : name(Name), contract(NULL) { }
    string getName() const { return(name); }
    void setContract(Contract *cn) { contract = cn; }
    Contract *getContract() const { return( contract); }
};
```

```
class Contract
{
    Person *pers; // association link to the class Person
    Company *comp; // association link to the class Company
    int contNum;
    int duration;
    static float rate;

public:
    Contract( Person* worker, Company* empl, int cN, int dr )
        : pers(worker), comp(empl), contNum(cN), duration(dr)
    {
        worker->setContract( this ); // set a link Person->Contract
        comp->setContract( this ); // set a link Company->Contract
    }

    string getPersonName() const { return(pers->getName()); }
    string getCompName() const { return(comp->getName()); }
    float getRate() const { return(rate); }
    int getDuration() const { return(duration); }
    int getContractNumber() const { return(contNum); }

};

float Contract::rate = 70.00;
```

```
int main()
{
    Person *worker = new Person( "John" );
    Company *company = new Company( "Bell Pty Ltd" );

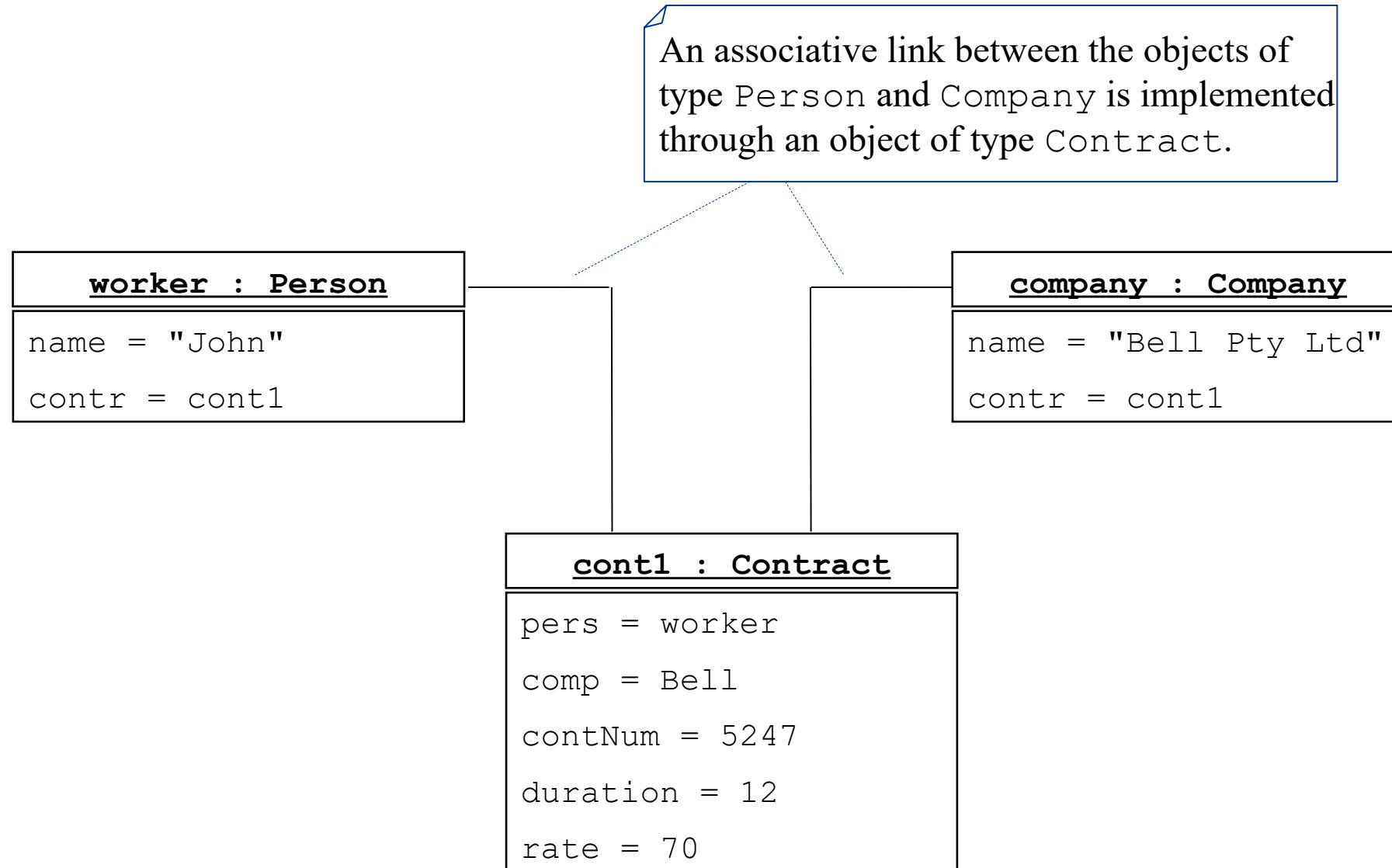
    Contract *cont1 = new Contract(worker, company, 5247, 12);

    cout << worker->getName() << " has a contract number "
    << worker->getContract()->getContractNumber()
    << " with " << worker->getContract()->getCompName()
    << endl;

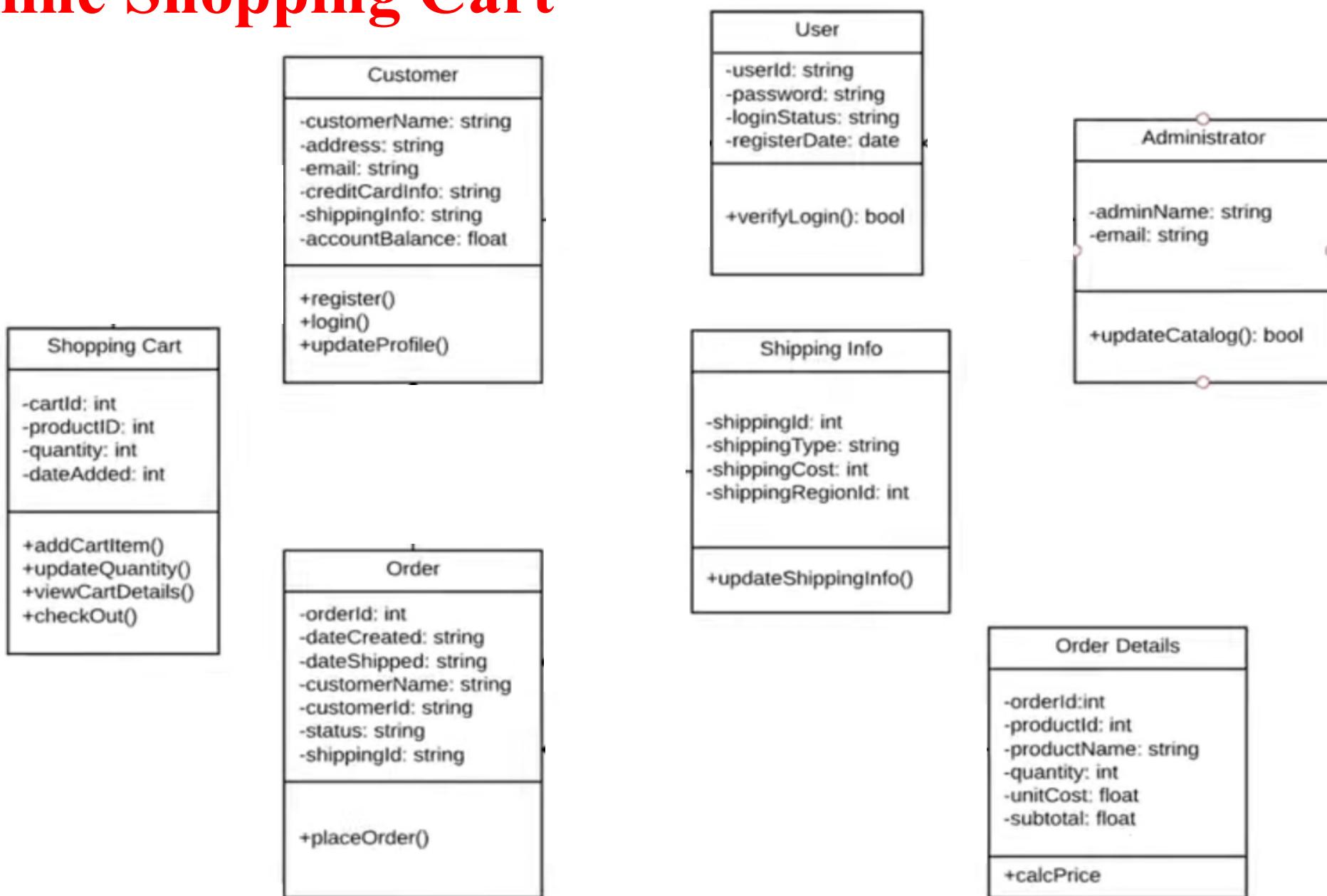
    cout << "Duration: "
    << worker->getContract()->getDuration()
    << " months" << endl;
    cout << "Rate: $" << worker->getContract()->getRate()
    << "/hr" << endl;
    return 0;
}
```

John has a contract number 5247 with Bell Pty Ltd
Duration: 12 months
Rate: 70 \$/hr

- Here goes an object diagram representing the relationship.



Online Shopping Cart



Online Shopping Cart

