Object Oriented Programming

Class Diagram

Relationship types

- Concepts does not exist in isolation. They coexist and interact.
- **Association** is a loose relationship in which objects of one class *"know"* about objects of another class (**has-a**)
- **Aggregation** is part-whole relationship (**is-part-of**)
- **Composition** is similar to aggregation, but is more strict (**contains**)
- **Inheritance** is a generalization-specialization relationship (**is-a**, **kind-of**)

- **Basic components of a class diagram**
- The standard class diagram is composed of three sections:
- **Upper section:** Contains the name of the class. This section is always required, whether you are talking about the classifier or an object.
- **Middle section:** Contains the attributes of the class. Use this section to describe the qualities of the class. This is only required when describing a specific instance of a class.
- **Bottom section:** Includes class operations (methods). Displayed in list format, each operation takes up its own line. The operations describe how a class interacts with data.

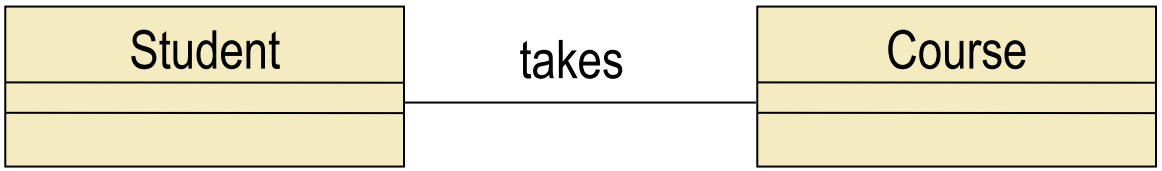
- **Basic components of a class diagram**
- In UML, a class represents an object or a set of objects that share a common structure and behaviour. They're represented by a rectangle that includes rows of the class name, its attributes, and its operations. When you draw a class in a class diagram, you're only required to fill out the top row—the others are optional if you'd like to provide more detail.
  - **Name:** The first row in a class shape.
  - **Attributes:** The second row in a class shape. Each attribute of the class is displayed on a separate line.
  - **Methods:** The third row in a class shape. Also known as operations, methods are displayed in list format with each operation on its own line.

- **Basic components of a class diagram**
- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations
- ~ denotes package attributes or operations

Basic components of a class diagram

Associations (I)

- **DEFINITION [Association] Association is a loose relationship in which objects of one class "*know*" about objects of the other class.**
- Association is identified by the phrase "**has a**".
- Association is a static relationship.
- Read relationships from left to right and from top to bottom

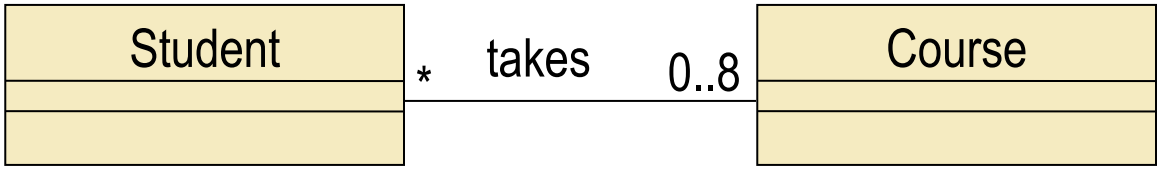


- Remarks: complicated to maintain => should be avoided as much as possible
- Implementation: member variables (pointers or references) to the associated object.

Associations (II)

- Multiplicity - The multiplicity applies to the adjacent class and is independent of the multiplicity on the other side of the association.

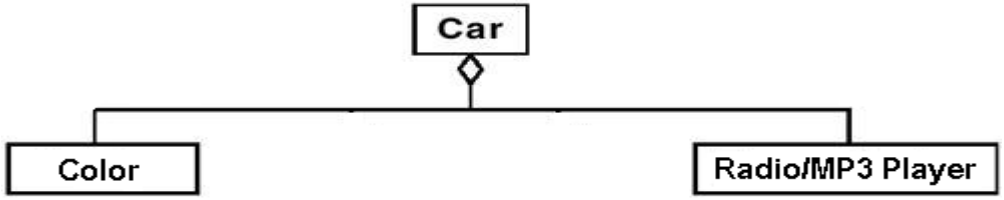
Notation	Meaning
1	Exactly one
*	Zero or many
0..5	Zero to five
0..4,6,10	Zero to five, six or ten
	Exactly one (default)



- Reflexive relationships: objects of the same class are related to each other
- n-ary associations:

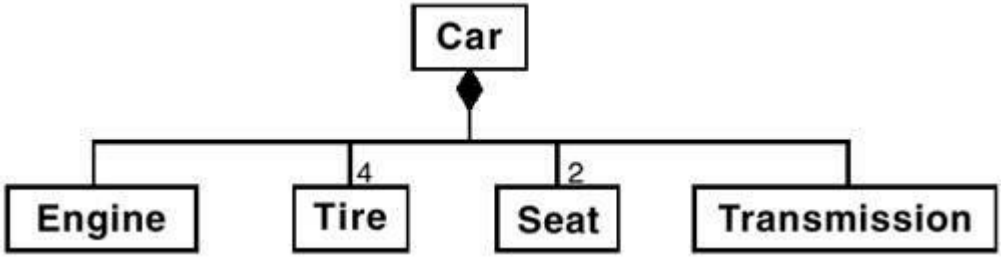
Aggregation

- Several definitions exist for aggregation and composition based on the following elements:
  - Accessibility: The part objects are only accessible through the whole object.
  - Lifetime: The part objects are destroyed when the whole object is destroyed.
  - Partitioning: The whole object is completely partitioned by part objects; it does not contain any state of its own.
- Both aggregation and composition represent a **whole-part** relationship.
- **DEFINITION [Aggregation] Aggregation is a relationship between part and whole in which:**
  - **parts may be independent of the whole;**
  - **parts may be shared between two whole instances.**
- Aggregation is identified by the phrase "**is-part-of**".
- Aggregation cannot be circular, i.e. an object cannot be part of itself.
- Example: A car has a color and a radio/mp3 player.



Composition

- **DEFINITION [Composition] Composition is a relationship between part and whole in which part may not be independent of the whole.**
- Composition is identified by the phrase "**contains**".
- **The contained object is destroyed once the container object is destroyed => No sharing between objects.**
- Stronger than aggregation.
- Example: A car contains an engine, four tires, two seats, and one transmission



Aggregation vs. Composition

- [Fowler, 2000, Page 85-87]

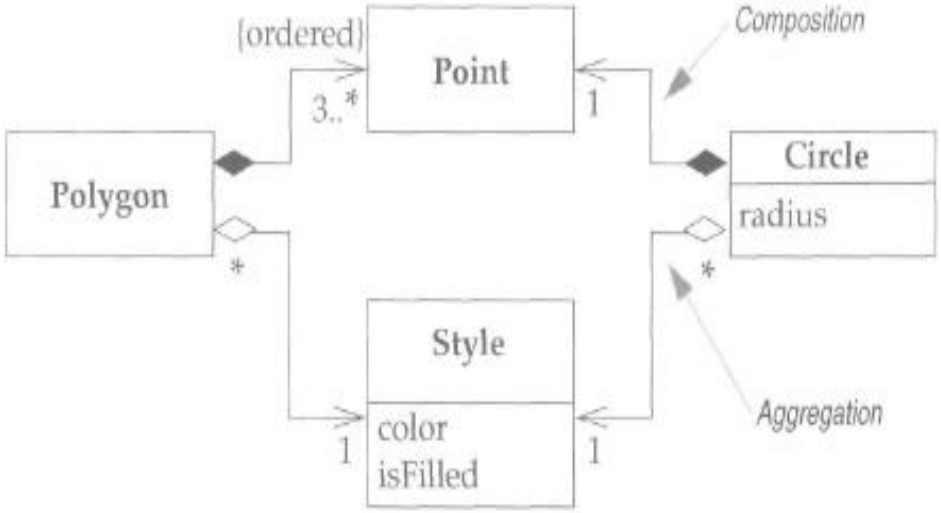


Figure 6-6: *Aggregation and Composition*

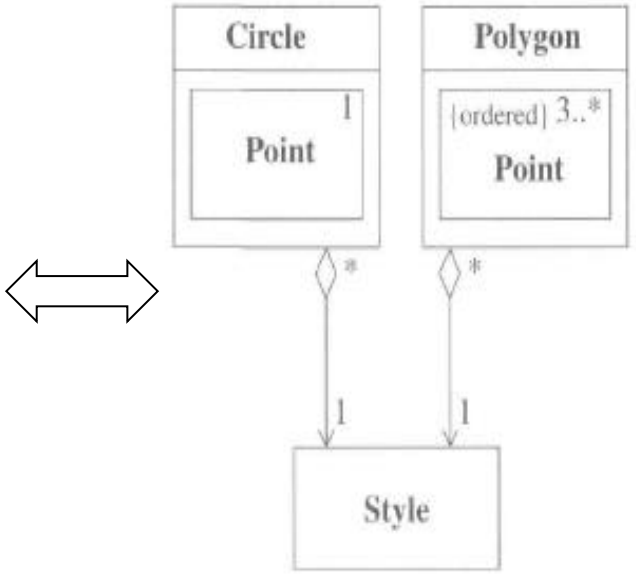
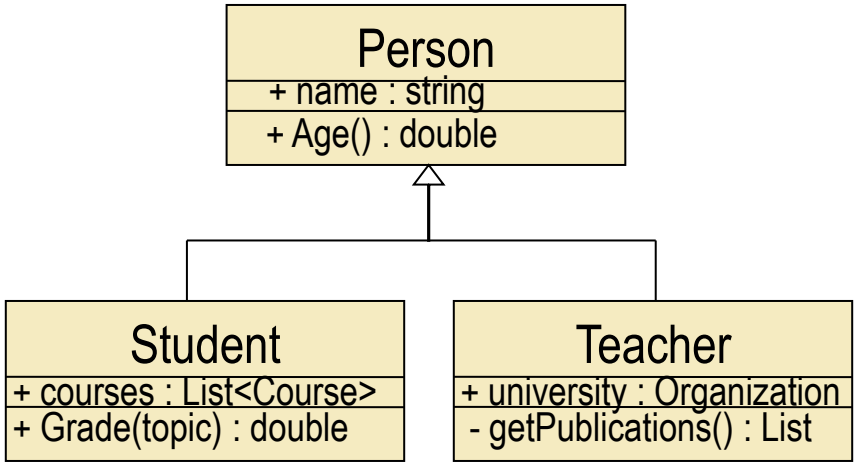


Figure 6-7: *Alternative Notation for Composition*

Differences between aggregation, composition and associations still under debate in software design communities.

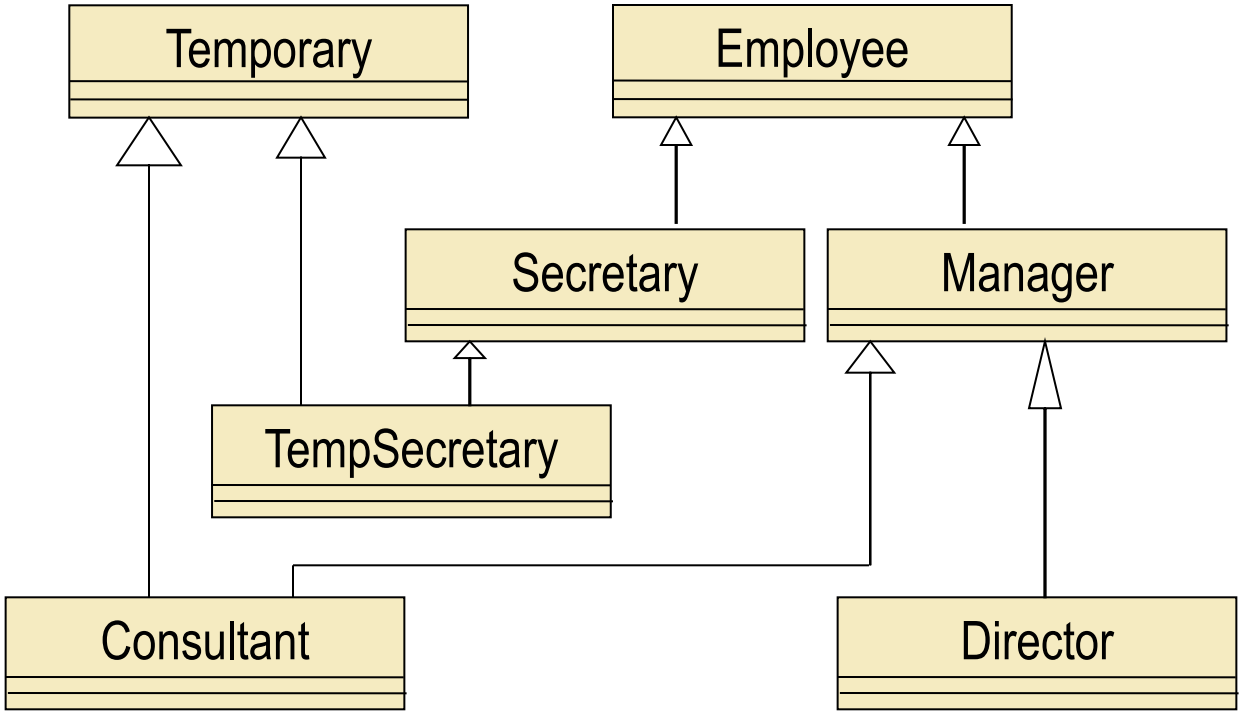
Inheritance

- **DEFINITION [Inheritance] Inheritance is a mechanism which allows a class A to inherit members (data and functions) of a class B. We say "A inherits from B". Objects of class A thus have access to members of class B without the need to redefine them.**
- Inheritance is identified by the phrase:
  - "**kind-of**" at class level (Circle is a kind-of Shape)
  - "**is-a**" at object level (The object circle1 is-a shape.)
- B is called superclass, supertype, base class or parent class.
- A is called subclass, subtype, derived class or child class.
- Introduced in Simula.
- Example:



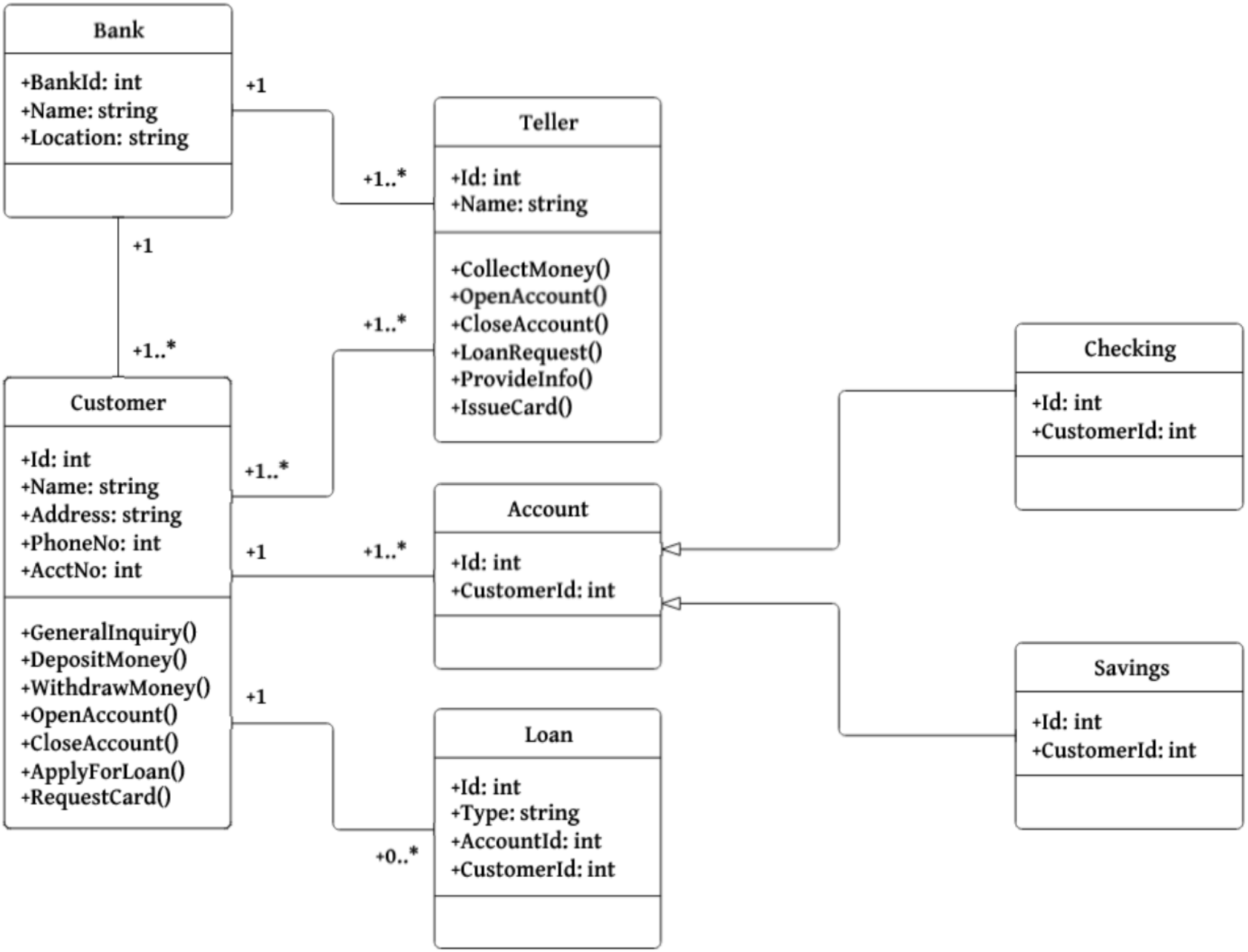
- **Inheritance graph / Class hierarchy**
- Multiple inheritance: a class has 2 or more parent classes.

Class Hierarchies



- HOMEWORK: Implement the hierarchy in C++ and override method *print* for all classes.

Class diagram for bank



Derived classes (I)

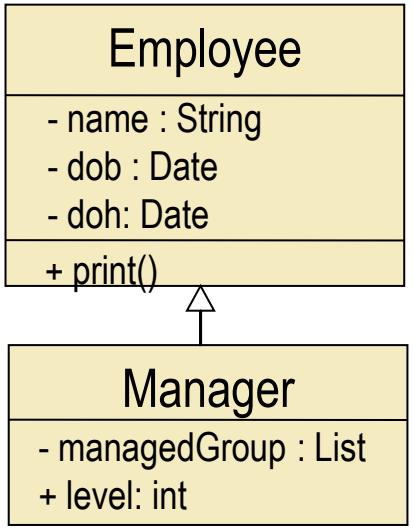
Do not multiply objects without necessity. (W. Occam)

- Inheritance is implemented in C++ through **derivation**.
- Derived classes does not have to know implementation details about base classes.
- Base classes are not “touched” in any way by inheritance.

```
class Employee {
public:
    Employee(String n, Date d);
    void print() const {
       cout << „Employee: ” << name << “ Dob: ”<< dob;
    }
private:
    String name;
    Date dob; // birth date
    Date doh; // hiring date
};

struct List {
    void add(Employee* );
};

// WITHOUT INHERITANCE
class Manager {
    Employee emp; // his/her properties as an employee
    List managedGroup;  // list of managed persons
};
```




```
class Manager : Employee {
public:
    Manager(const char* name);
    int level;
private:
    list managedGroup;  // list of managed persons
};
```

Derived classes (II)

- Syntax:

class DerivedClass : [access modifier] BaseClass { …. } ;

- A class must be declared in order to be used as a base.

- As a derived class is a **kind-of** base class => a derived class object (e.g. Manager) can be used wherever a base class is acceptable (e.g. Employee). But, not the other way around!

- Example:

```
void f(Manager& m, Employee& e) {
    Employee* pe = &m; // correct: every Manager is an Employee
    Employee& re = m;  // correct

    List l;
    l.add(&m);
    l.add(&e);
    l.add(new Manager("John Doe"));
    l.add(new Employee("Vasile Popescu", Date(10,10,1970)));

    g(m); // ok
    g(Employee("Vasile Popescu", Date(10,10,1970)));
}
```

```
void g(Employee& e) {
    Manager* pm = &e; // error: not every
                      // Employee is a Manager
    // brute-force
    pm = static_cast<Manager*>(&e);

    cout << pm->level;
}
```

Error at runtime due to static_cast conversion in g implementation!

Access control (I)

- Member functions of derived class have access to public and protected members of base class, but not to private ones.
- To control the access to inherited members from base class, access control specifiers are used.
- Example:

```
class Manager : public Employee { /* declarations */};

class Manager : protected Employee { /* declarations */};

class Manager : private Employee { /* declarations */};
```

- If missing, private is considered.

Base class	Access control	Type in derived class	External access
private protected public	private private private	not accessible private private	not accessible not accessible not accessible
private protected public	protected protected protected	not accessible protected protected	not accessible not accessible not accessible
private protected public	public public public	not accessible protected public	not accessible not accessible accessible

Access control (II)

```
class B {
    int x;
protected:
    int y;
public:
    int z;
};

class A : B {
    void f() {
        x = 10; // ERROR: private
        y = 20; // CORRECT!
        z = 30; // CORRECT!
    }
};

class AA : public B {
};

class AAA : AA {
    void aaa();
};
```

```
void f() {
    B b;
    A a;
    AA aa;
    b.x = 10;  // ERROR: private
    b.y = 20;  // ERROR : protected
    b.z = 30;  // CORRECT!
    a.x = 10;  // ERROR: private
    a.y = 20;  // ERROR: private
    a.z = 30;  // ERROR: private
    aa.x = 10; // ERROR: private
    aa.y = 20; // ERROR: protected
    aa.z = 30; // CORRECT!
}

void AAA::aaa() {
    x = 10; // ??
    y = 20; // ??
    z = 30; // ??
}
```

- **Keep in mind: Data hiding (encapsulation) is a key principle in OOP! => Try to minimize the number of functions that have access to members.**

- Strange things happens? Really?... What is going on here???

```
a.z = 50; // ERROR: z private in A

B* pb = &a;

pb->z = 50; // OK: z public in B
```

Constructors and destructor

- An instance of a derived class contains an instance of a base class => need to be initialized using a constructor
- Use initialization list to call the appropriate constructor

```
class Manager : public Employee {
    list managedGroup;
    int level;
public:
    Manager(const String& s, const Date& d, List &g)
      : Employee(s, d),
        managedGroup(g) , level (0){
    }
};
```

- "Slicing"

```
void f() {
    List l;
    Manager m(„Popescu Vasile", Date(04, 09, 1965), l);
    Employee c = m; // only Employee part is copied
}
```

- Objects are constructed from the bottom to up: 1) base, 2) non-static data members, 3) derived class.
- Example:
  - allocate memory to hold a Manager instance
  - call Employee(s,d) to initialize base class
  - call List(g) to initialize managedGroup member
  - execute Manager(…) body
- Objects are destroyed in the reverse order: derived class, non-static data members, base.

Functions with the same prototype

- A member function of derived class may have the same prototype with the function from the base class.
- Base class function is not impacted, being still accessible using name resolution operator
- Example:

```
class Manager : public Employee {
    list managedGroup;
     int level;
public:
    Manager(String s1, String s2, List &g)
       : Employee(s1, s2), managedGroup(g) {
    }

    void print() const {
        Employee::print(); // OK: call base class member
        cout << "Managed group: " << managedGroup;
         cout << "Level:" << level;
    }
};
```

```
void f()  {
   Manager m(„Popescu", Date(07, 09, 1978), List());
   Employee* pa = &m;
   m.print(); // Manager::print()
   m.Employee::print(); // base's print
   pa->print(); // Employee::print
}
```

Virtual functions (I)

- Two solutions for invoking the appropriate behavior:
  - Type fields
  - Virtual functions
- **DEFINITION [Virtual function, method] A function that can be redefined in each derived class is called virtual function (or method).**
- **The prototype of redefined function must have the same name and the same list of arguments, and can only slightly differ in return value.**
- Syntax: **virtual <function prototype>;**

• The compiler will ensure that the right v.f. is invoked for each object.

• Example:

```
class Employee {
public:
    virtual void print() const {
        cout << „Employee: ” << name << “ Dob: ”<< dob;
    }
    // other declarations
};
```

```
class Manager : public Employee {
public:
    void print() const {
        Employee::print(); // OK: call base class member
        cout << “Managed group: ” << managedGroup;
    }
};

void f()  {
    Manager m(„Popescu”, Date(07, 09, 1978), List());
    Employee* pe = &m;
    pe->print();   // in fact, Manager::print
    pe = new Employee(…);
    pe->print(); // this time, Employee::print
}
```

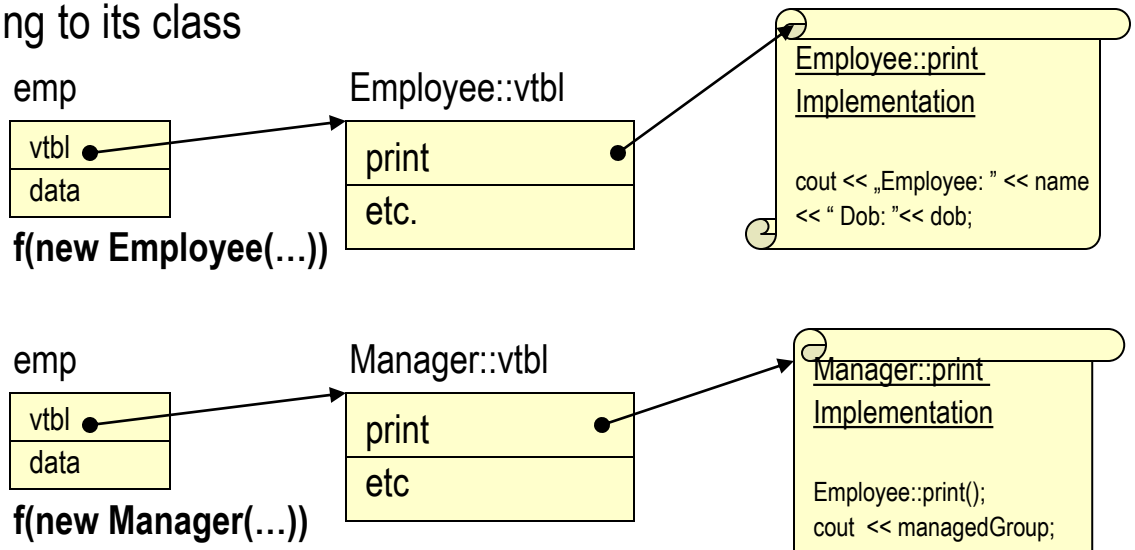
Virtual functions (II)

- **DEFINITION [Overriding] A function from a derived class with the same name and the same list of arguments as a virtual function in a base class is said to override the base class version of the virtual function.**
- If a virtual function is not overrode in a derived class, then the base class implementation is used.
- Q: How the correspondence between object and proper (virtual) function is stored?
- A: Each instance (object) of a class having v.f. holds a pointer to the VFT (**Virtual Functions Table**) corresponding to its class
- Example:

```
void f(Employee* emp) {
    emp->print();
}
```

⇩

```
JMP emp->vtbl->get('print')
```



Polymorphism (I)

- **DEFINITION [Polymorphism] Getting "the right" behavior from base class objects independently of exactly what kind of instance (base of various derived classes) is actually used is called polymorphism.**
- **DEFINITION [Polymorphic type] A type/class with virtual functions is called polymorphic type.**
- To get polymorphic behavior, objects must be manipulated through pointers or references, otherwise no run-time polymorphism is used.
- Static binding – at compile time. Examples:
  - Employee e; e.print();
  - Employee* pe; pe->Manager::print();
- Dynamic binding – at run-time. Examples:
  - Employee* pe = new Manager(); pe->print();

Polymorphism (II)

- Example:

```
class HeadOfDepartment : public Employee {
    int departmentID;
public:
    HeadOfDepartment (String& s, Date& d, int id)
        : Employee(s, d), departmentID(id) {
    }

    void print() const {
        Employee::print();
        cout << „Department: ” << departamentID;
    }
};
```

```
// polymorphic behavior
void printList(List& lista) {
    for(int i=0; i<lista.size(); i++)
        lista.get(i)->print(); // right behavior is invoked
}

int main(int, char*[]) {
    List l;
    l.add(new Manager(„Popescu”, Date(01,01,1968), List()));
    l.add(new HeadOfDepartment(„Alexandrescu”, Date(), 1001));
    l.add(new Employee(“Ionescu”, Date(10,10,1970));
    printList(l);
}
```

Abstract classes

- **DEFINITION [Pure v.f.] A pure virtual function is a virtual function declared, but not implemented in a base class.**
- A pure virtual function has to be override by all derived classes; otherwise it remains pure.
- **DEFINITION [Abstract class, Interface] A class having at least one pure v.f. is called abstract class.**
- Abstract classes cannot be instantiate (it's an incomplete type).
- Example

```
class Shape { // abstract class
    public:
        virtual void draw() const = 0; // pure virtual
};

class Circle  : public Shape { // concrete type
    public:
         void draw() const;
}
```

```
void Circle::draw() const {
    cout << "Draw the circle;";
}

void f()  {
    Shape sh; // ERROR: Impossible to instantiate abstract classes
    Shape* sh = new Circle; // ok
    sh->draw(); // Circle::draw
}
```

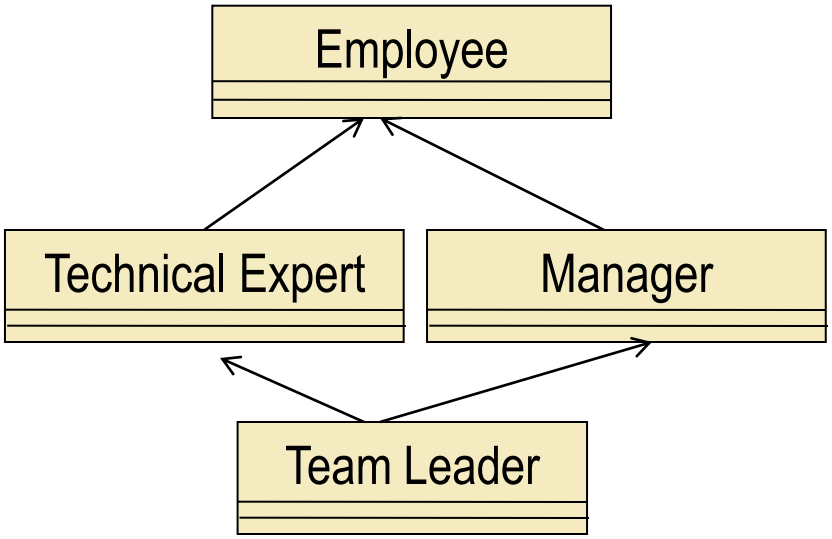
Multiple inheritance (I)

- **DEFINITION [Multiple inheritance] Multiple inheritance is when a class inherits characteristics from two or more base classes.**
- Increased flexibility of class hierarchies => complex hierarchies (graph-like hierarchies)
- Example:

```
class Temporary {
    Date start, end; // period of collaboration
}

// Multiple inheritance
class Consultant : protected Temporary, public Manager {
};
```

- Diamond-like inheritance graph



Multiple inheritance (II). Virtual base class

- Problems: a base class (Employee) may be included twice in a derived class (diamond-like inheritance)
  - memory wasting
  - how to access members from a base class that is included twice (Employee)?
- **DEFINITION [Virtual base class] If a class is declared as virtual base, then in a diamond-like inheritance its instance is created and initialized only once.**
- Syntax:

  class DerivedClass : [private|protected|public] virtual BaseClass { /* decl. */ };
- Example:

```
class Base {
};

class Derived1 : virtual public Base {
};

class Derived2 : virtual public Base {
};

class Derived : public Derived1, public Derived 2 {
    public:
        Derived() : Base(), Derived1(), Derived2() { }
};
```

- Need to explicitly call the virtual base class constructor
- Steps of object initialization:
  - call virtual base constructor
  - call constructors of base classes in order of their declaration
  - initialize derived class members
  - initialize derived object itself

```
void f() {
    Derived d;
}
```