Unit 7

Agenda

Inheritance

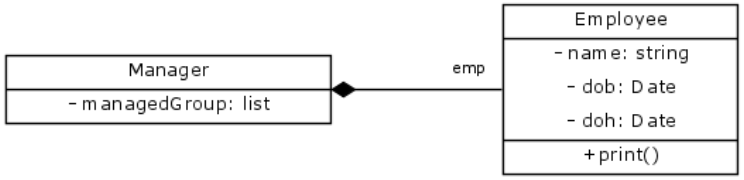
- Definition
- Derived classes
- Access control
- Constructors and destructor
- Virtual functions
- Polymorphism
- Multiple inheritance

Manager is an Employee

In an organization, each Manager is an Employee as well.

Let's model this using Composition.

```
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* );
};

// WITH COMPOSITION
class Manager {
     Employee emp; // his/her properties as an employee
     list managedGroup;  // list of managed persons
};
```



Although technically it is possible, it is not inline with the meaning of the relationship between the 2 concepts; this is not a whole-part relationship, but a specialization.

We need something else….

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.

Introduced in Simula.

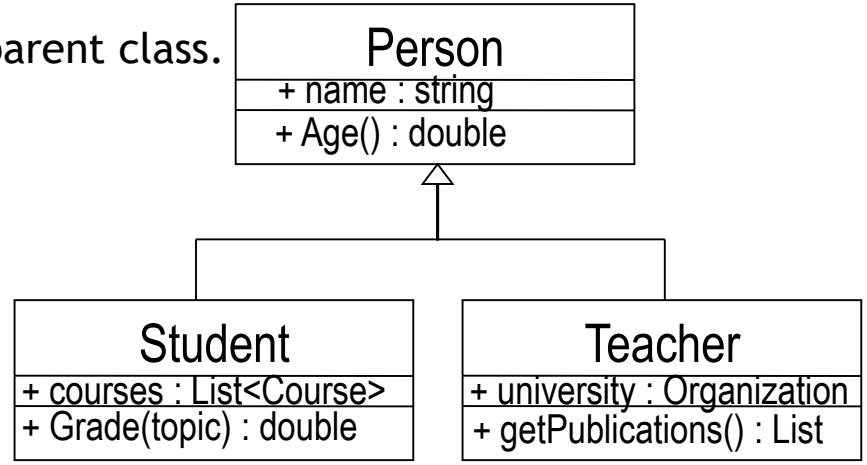
Inheritance is identified by the phrase:

- "**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.

Inheritance graph / Class hierarchy



Derived classes

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

Inheritance is implemented in C++ through **derivation**.

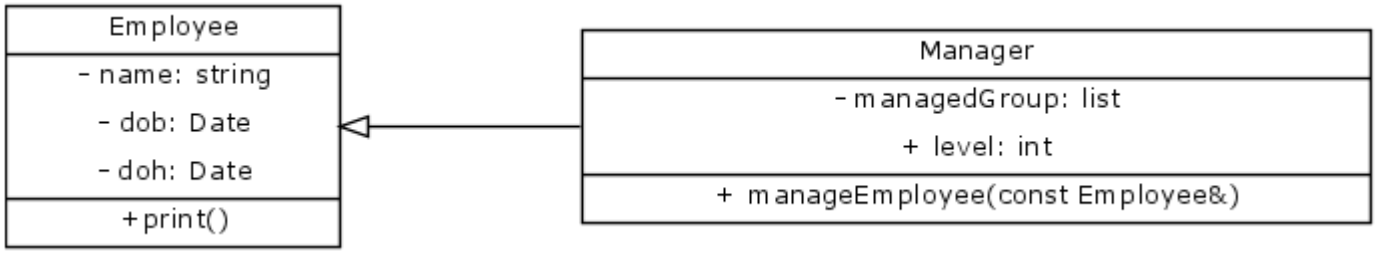
Derived classes does not have to know implementation details about base classes.

Base classes are not “touched” in any way by inheritance.

```
class Manager : public Employee {
public:
     Manager(const char* name);
     int level;

private:
     // list of managed persons
     list managedGroup;
};
```

Syntax

```
class DerivedClass :
  [access_modifier] BaseClass {

} ;
```



Up-casting and down-casting

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) *(up-casting)*

But, not the other way around! *(down-casting)*

```
void f(Manager& m, Employee& e) {
     Employee* pe = &m; // correct: every Manager is an Employee
     Employee& re = m;  // correct
     Manager* pm  = &e; // error: not every Employee is a Manager

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

     // Next is ok!
     g(m);

     // Next call generates a runtime exception because a non-Manager
     // instance is passed-in as actual argument
     g(Employee{“Vasile Popescu”, Date{10,10,1970}});
}
```

```
void g(Employee& e) {
    // dangerous brute-force
    Manager* pm = static_cast<Manager*>(&e);

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

Access control (I)

ALL members of the base class are "inherited" in the derived class, but not all are accessible.

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 modifiers (private, protected, public) are used.

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

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

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

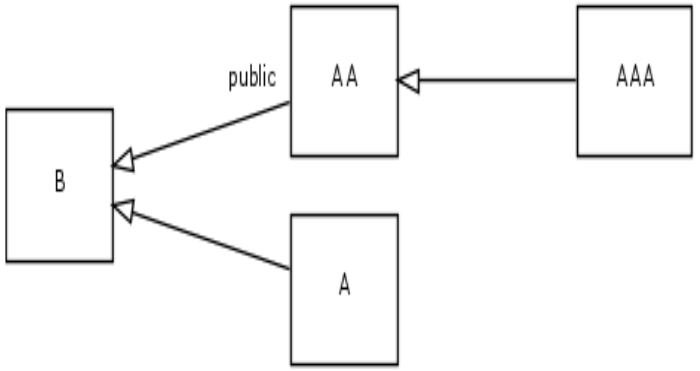
If missing, private is considered for a type (class) declared using **class** keyword and public if **struct** is used.

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

Remark: Because protected data member are accessible to derived classes member functions, they are likely to create maintenance issues in the future. Their usage might be considered a design error.

Access control (II)

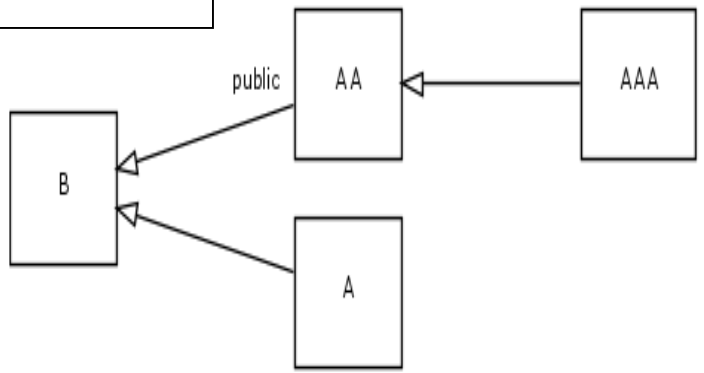
```
class B {
     int x;
protected:
     int y;
public:
     int z;
};

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

class AA : public B {
};

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



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();
};
```

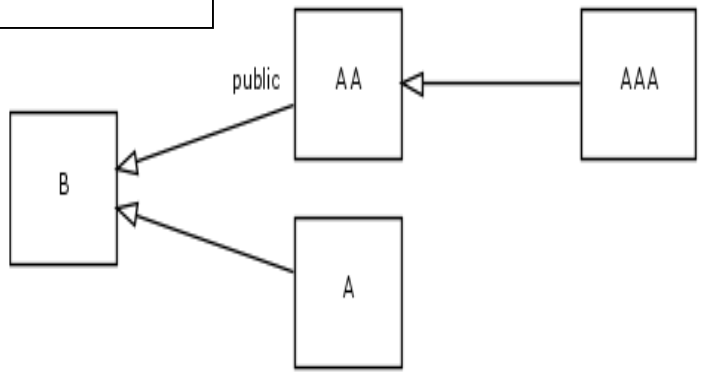
```
void f() {
     B b;
     A a;
     AA aa;
     b.x = 10;
     b.y = 20;
     b.z = 30;
     a.x = 10;
     a.y = 20;
     a.z = 30;
     aa.x = 10;
     aa.y = 20;
     aa.z = 30;
}

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



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();
};
```

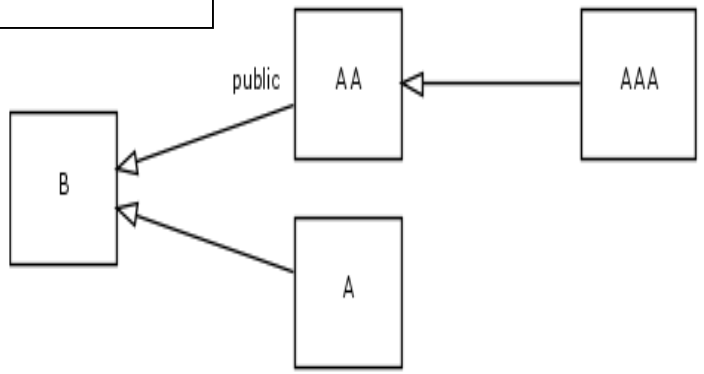
```
void f() {
     B b;
     A a;
     AA aa;
     b.x = 10;  // ERROR: private
     b.y = 20;  // ERROR: protected
     b.z = 30;  // CORRECT!
     a.x = 10;
     a.y = 20;
     a.z = 30;
     aa.x = 10;
     aa.y = 20;
     aa.z = 30;
}

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



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();
};
```

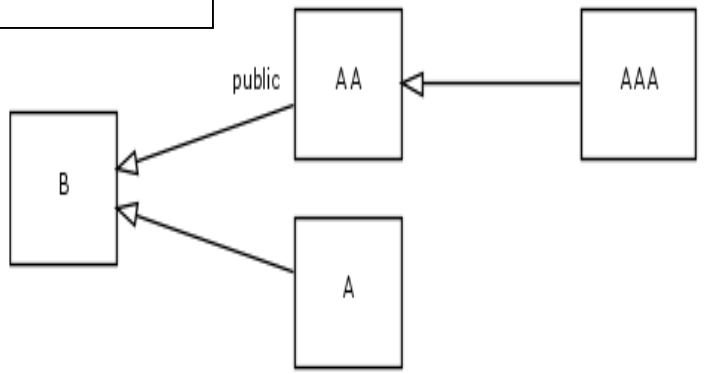
```
void f() {
     B b;
     A a;
     AA aa;
     b.x = 10;  // ERROR: private
     b.y = 20;  // ERROR: protected
     b.z = 30;  // CORRECT!
     a.x = 10;  // ERROR: private
     a.y = 20;  // ERROR: private
     a.z = 30;  // ERROR: private
     aa.x = 10;
     aa.y = 20;
     aa.z = 30;
}

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



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();
};
```

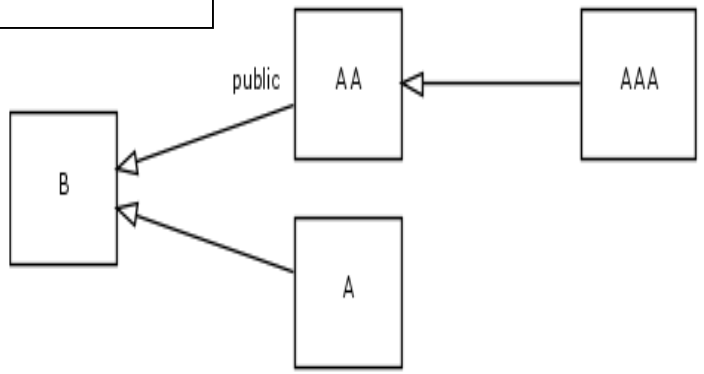
```
void f() {
     B b;
     A a;
     AA aa;
     b.x = 10;  // ERROR: private
     b.y = 20;  // ERROR: protected
     b.z = 30;  // CORRECT!
     a.x = 10;  // ERROR: private
     a.y = 20;  // ERROR: private
     a.z = 30;  // ERROR: private
     aa.x = 10; // ERROR: private
     aa.y = 20; // ERROR: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

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



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; // ERROR: private
     y = 20; // CORRECT: protected
     z = 30; // CORRECT: public
}
```



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();
};
```

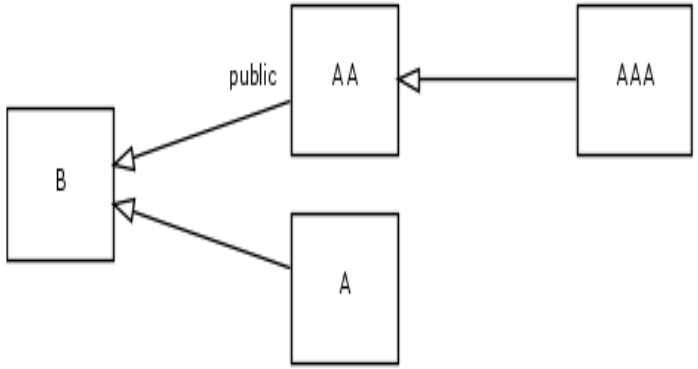
```
void f() {
     B b;
     A a;
     AA aa;
     b.x = 10;  // ERROR: private
     b.y = 20;  // ERROR: protected
     b.z = 30;  // CORRECT!
     a.x = 10;  // ERROR: private
     a.y = 20;  // ERROR: private
     a.z = 30;  // ERROR: private
     aa.x = 10; // ERROR: private
     aa.y = 20; // ERROR: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

void AAA::aaa() {
     x = 10; // ERROR: private
     y = 20; // CORRECT: protected
     z = 30; // CORRECT: public
}
```

Keep in mind: Data hiding (encapsulation) is a key principle to OOP! =>

Try to minimize the number of functions that have access to members.

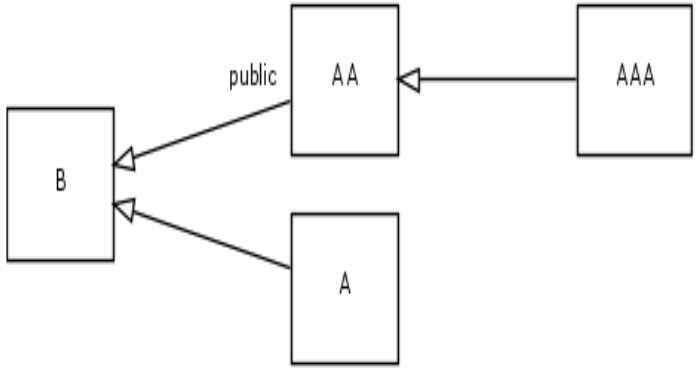
Access control (III)

```
A a;

a.z = 50;

B* pb = &a;

pb->z = 50;
```



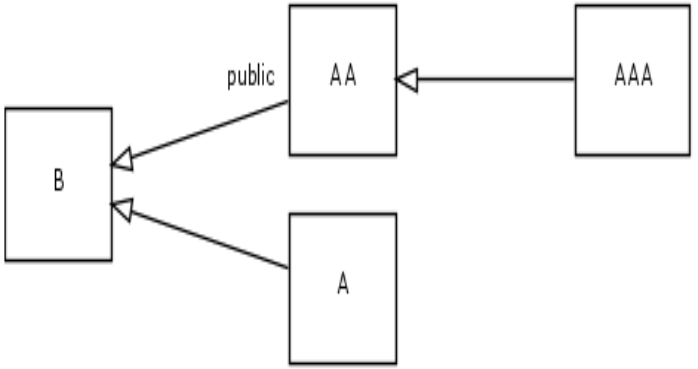
Access control (III)

```
A a;

a.z = 50; // ERROR: z is private

B* pb = &a;

pb->z = 50; // CORRECT: z is public
```



Access control (III)

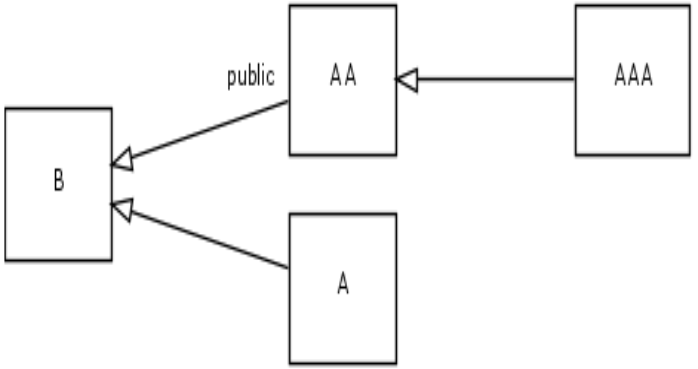
```
A a;

a.z = 50; // ERROR: z is private

B* pb = &a;

pb->z = 50; // CORRECT: z is public
```



Ooops! Is **z** accessible or not?

Access control (III)

```
A a;

a.z = 50; // ERROR: z is private

B* pb = &a;

pb->z = 50; // CORRECT: z is public
```



Ooops! Is **z** accessible or not?

z should not be accessible (as per the design of class A that specifies B as private (hidden) base) and IT IS NOT ACCESSIBLE!

The statement

```
B* pb = &a;
```

is erroneous and flagged appropriately by the compiler because

B is not a public base class of A.

Access control in a nutshell

- If B is a private base, its public and protected members can be used only by member functions and friends of D. Only friends and members of D can convert a D* to a B*.

- If B is a protected base, its public and protected members can be used only by member functions and friends of D and by member functions and friends of classes derived from D. Only friends and members of D and friends and members of classes derived from D can convert a D* to a B*.

- If B is a public base, its public members can be used by any function. In addition, its protected members can be used by members and friends of D and members and friends of classes derived from D. Any function can convert a D* to a 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} {
     }
};
```

```
Manager myGoodBoss;
```

Objects are constructed from the bottom to up:

1) base,
2) non-static data members,
3) derived class.

Objects are destroyed in the reverse order: derived class, non-static data members, base.

Constructing an instance of class Manager breaks down to the following steps:

(1) allocate memory to hold a Manager instance
(2) call Employee(s, d) to initialize base class
(3) call List(g) to initialize managedGroup member
(4) execute Manager(…) constructor body

Slicing

```
void f() {
    List l;

    Manager m{„Popescu Vasile”, Date{04, 09, 1965}, l};

    Employee c = m; // only Employee part is copied
}
```

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

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

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

```
class Employee {
public:
     Employee(String n, Date d);
     void print() const;

private:
     String name;
     Date dob; // birth date
     Date doh; // hiring date
};


void Employee::print() const {
  cout << "Employee: " << name
       << " Dob: "<< dob;
}
```

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

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

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

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

```
class Employee {
public:
     Employee(String n, Date d);
     void print() const;

private:
     String name;
     Date dob; // birth date
     Date doh; // hiring date
};


void Employee::print() const {
  cout << "Employee: " << name
       << " Dob: "<< dob;
}
```

```
void main()  {
    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
}
```

Exercise: Invoking expected behavior

Make any necessary changes in the previous example so that pa->print() call ‘re-directs’ to the behavior of print from Manager class.

Exercise: Invoking expected behavior

Make any necessary changes in the previous example so that pa->print() call ‘re-directs’ to the behavior of print from Manager class.

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

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

```
Employee::Employee(String n, Date d, int t)
     : name{n}, dob{d}, type{t}
{
     // other initializations
}
```

```
class Employee {
public:
     static const int TMANAGER = 10;

     Employee(String n, Date d, int t);
     void print() const;

private:
     String name;
     Date dob; // birth date
     Date doh; // hiring date
     int type; // type of Employee
};

void Employee::print() const {
     cout << ”Employee: ” << name
          << “ Dob: ”<< dob;

     switch(type) {
          case Employee::TMANAGER:
          ((Manager*)this)->print();
          break;
     }
}
```

Virtual functions (I)

Two solutions for invoking the appropriate behavior:

- Type fields (see previous slide)
- Virtual functions

Syntax
`virtual <function prototype>;`

DEFINITION [Virtual function, method] A function that can be redefined in each derived class is called virtual function (or method).

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.

The prototype of redefined function must have the same name and the same list of arguments, and can only slightly differ in return value.

By default, a function that overrides a virtual function itself becomes virtual.

The compiler will ensure that the right virtual function is invoked for each object.

If a virtual function is not overrode in a derived class, then the base class implementation is used.

Virtual functions (II)

Without virtual function print.

```
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(); // call base class member
          cout << “Managed group: ” << managedGroup;
          cout << “Level:” << level;
     }
};
```

```
class Employee {
public:
     Employee(String n, Date d);
     void print() const;

private:
     String name;
     Date dob; // birth date
     Date doh; // hiring date
};


void Employee::print() const {
  cout << „Employee: ” << name
       << “ Dob: ”<< dob;
}
```

```
void main()  {
    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 (II)

With virtual function print.

```
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(); // call base class member
          cout << "Managed group: " << managedGroup;
          cout << "Level:" << level;
     }
};
```

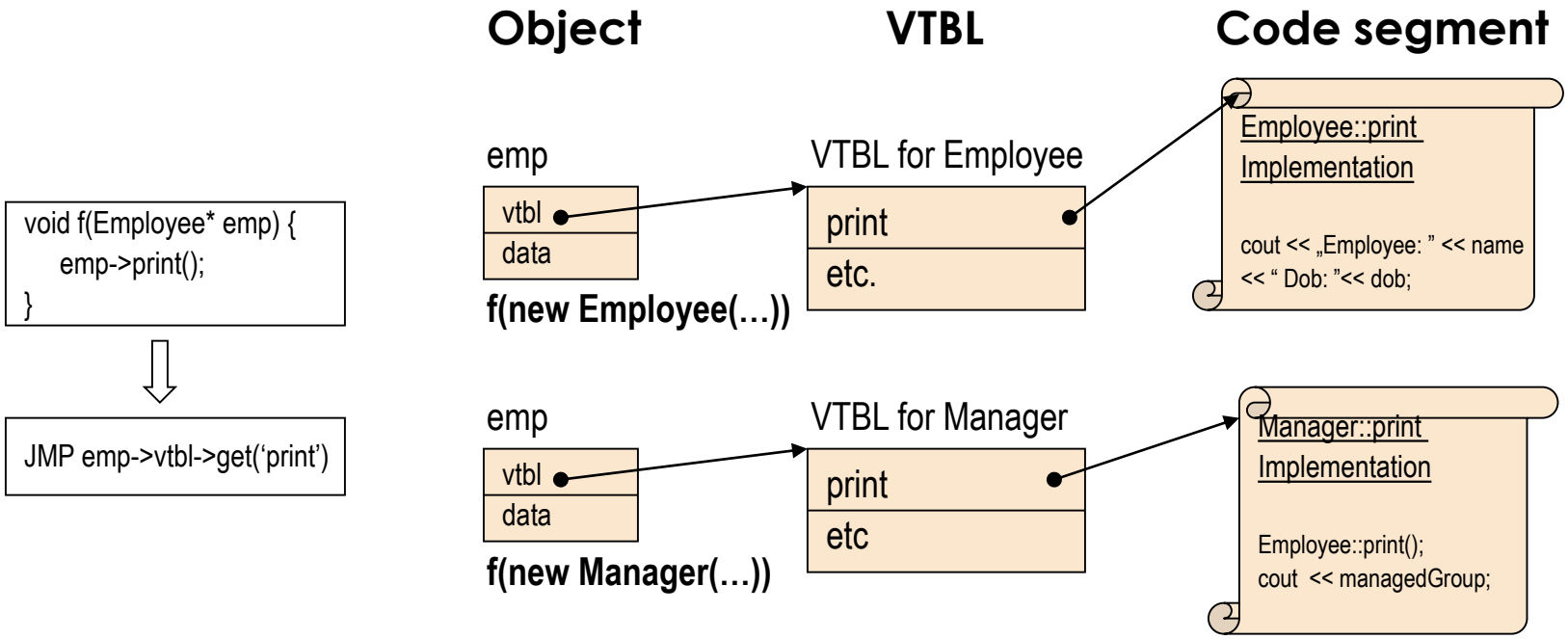
```
class Employee {
public:
     Employee(String n, Date d);
     virtual void print() const;

private:
     String name;
     Date dob; // birth date
     Date doh; // hiring date
};


void Employee::print() const {
  cout << „Employee: " << name
       << " Dob: "<< dob;
}
```

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

Virtual functions (III)

Q: How the correspondence between object and proper (virtual) function is achieved?

- Each instance (object) of a class having virtual functions holds a pointer to a VTBL (**Virtual Functions Table**) corresponding to its class.
- The VTBL has an entry of type <virtual_function_name, address>
- Indirection



Polymorphism (I)

DEFINITION [Polymorphism] Getting "the right" behavior from base class functions 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 (run-time) polymorphic type.

To get polymorphic behavior, objects must be manipulated through **pointers or references** and the member functions must be **virtual**, otherwise no run-time polymorphism is used.

Static binding – at compile time. Examples:

- `Employee e; e.print(); // by type`
- `Employee& re = . . .; re.Manager::print(); // f. q. n`
- `Employee* pe; pe->Manager::print(); // fully qualified name`

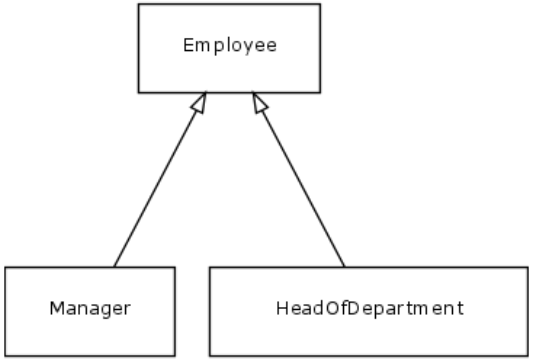
Dynamic binding – at run-time. Examples:

- `Employee* pe = new Manager(); pe->print(); // polymorphism`
- `Employee& re = m; re.print(); // polymorphism`

Polymorphism (II)

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

int main(int, char*[]) {
     List l; // don’t worry too much about the type List ☺
     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);
     return 0;
}
```



Employee and Manager classes are defined in previous slides.

Constructors and destructor

A virtual function invoked from a constructor or a destructor reflects that the object is partially constructed / destroyed => It is therefore typically a bad idea to call a virtual function from a constructor or a destructor.

override specifier

In a large or complicated class hierarchy with many **virtual** functions, it is best to use virtual only to introduce a new virtual function and to use **override** on all functions intended as overriders. Using override is a bit verbose but clarifies the programmer's intent.

override is a suffix – it always come last in a declaration! Do NOT repeat it in function implementation

override is not a keyword, it is a *contextual keyword*.

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

     void print() const override ;
};

void print() const override { // ERROR override repeated in impl.
    Employee::print();
    cout << „Department: ” << departamentID;
}
```

```
int override = 7; // OK!

// OK as well
class C : public B {
  int override;
  int f() override {
      return override +
             ::override;
  }
};
```

final specifier

final prevents further overriding of a virtual function.

Used in class declaration makes all virtual functions final, so that derived classes cannot override the behaviour.

final on the class not only prevents overriding, it also prevents further derivation from a class

```
class HeadOfDepartment : public Employee {
     // ...

     void print() const override final;
};

void print() const final { // ERROR final repeated in impl.
    Employee::print();
    cout << „Department: ” << departamentID;
}

class HeadOfDepartment final : public Employee {
     // ...
}
```

```
int final = 7; // OK!

// OK as well
class C : public B {
  int final;
  int f() override final {
      return final  +
             ::final;
  }
};
```

using keyword

```
struct Base {
    void f(int);
};

struct Derived : Base {
     void f(double);
};

void use(Derived d) {
    d.f(1); // call Derived::f(double)
    Base& br = d;
    br.f(1);// call Base::f(int)
}
```

using keyword

```
struct Base {
    void f(int);
};

struct Derived : Base {
     void f(double);
};

void use(Derived d) {
    d.f(1); // call Derived::f(double)
    Base& br = d;
    br.f(1);// call Base::f(int)
}
```

```
struct D2 : Base {
    using Base::f; // bring all fs from Base into D2
    void f(double);
};

void use2(D2 d) {
   d.f(1); // call D2::f(int), that is, Base::f(int)
   Base& br = d;
   br.f(1); // call Base::f(int)
}
```

- using declarations can be used to add a function to a scope.
- Multiple **using** declarations can bring functions from different basis,
- using cannot be used to gain access to private members

```
struct B1 {
    void f(int);
};

struct B2 {
    void f(double);
};

struct D : B1, B2 {
    using B1::f;
    using B2::f;
    void f(char);
};
```

```
void use(D d)
{
  d.f(1); // call D::f(int), that is, B1::f(int)
  d.f('a'); // call D::f(char)
  d.f(1.0); // call D::f(double), that is, B2::f(double)
}
```

Return type relaxation. Covariance

Covariant return rule: if the original return type of a virtual function was **B***, then the return type of the overriding function may be **D***, provided **B** is a public base of **D**. Similarly, a return type of **B&** may be relaxed to **D&**.

```
struct Base {
    virtual char* describe() = 0;
};

struct Derived : public Base {
    char* describe() override { return "Derived"; }
};

struct Factory {
    virtual Base* create() = 0;
};

struct DerivedFactory : Factory {
    Derived* create() override {
        return new Derived( );
    }
};

void use(Factory *f) {
      Base* pb = f->create();
}
```

The relationship between Derived and Base is **covariant** with the relationship between Factory and DerivedFactory.

DEFINITION [virtual constructor] Member function such create is sometimes called virtual constructor because they are used to indirectly create objects.

Contravariance

```
struct Animal {
    virtual char* describe() = 0;
};

struct Cat : public Animal {
    char* describe() override { return “I’m a cat\n”; }
};

struct Mouse : public Animal { /* impl of describe() */ }

struct CatDoctor {
    virtual void treat(Cat& cat) {
        std::cout << “CatDoctor::treat” << cat.describe() << std::endl;
    }
};

struct AnimalDoctor : public CatDoctor { // AnimalDoctor is a CatDoctor, right?
    // Does not compile due to override keyword
    // Does not override Base::treat
    void treat(Animal& anml) override {
        std::cout << “AnimalDoctor::treat” << anml.describe() << std::endl;
    }
};

int main() {
    Cat tom;
    Mouse jerry;

    // Assume that override keyword is not used in the definition of AnimalDoctor
    AnimalDoctor ad;
    ad.treat(tom);  // calls AnimalDoctor::treat
    ad.treat(jerry);// calls AnimalDoctor::treat
    ad.CatDoctor::treat(tom); // force calling CatDoctor::treat
}
```

The relationship between Animal and Cat is **contravariant** with the relationship between AnimalDoctor and CatDoctor: an AnimalDoctor IS-A CatDoctor precisely because a Cat IS-AN Animal.

Remark: the derived treat function accepts the broader type (Animal) because it must do at least as much as the base's function is able to do.

C++ does not support contravariance.

Pure virtual functions. Abstract classes

DEFINITION [Pure virtual function] A pure virtual function is a virtual function declared, but not implemented in the base class.

A pure virtual function has to be override by all derived classes; otherwise it remains pure.

DEFINITION [Abstract class] A class having at least one pure virtual function is called abstract class.

An abstract class cannot be instantiated (it is an incomplete type).

```
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;";
}
```

Syntax

```
virtual <function prototype> = 0;
```

Remark: In VTBL, a pure virtual function's entry is associated value 0.

```
void foo()  {
     Shape sh; // ERROR: Impossible to instantiate abstract classes!
     Shape* sh = new Circle; // OK: Concrete class
     sh->draw(); // Circle::draw
}
```

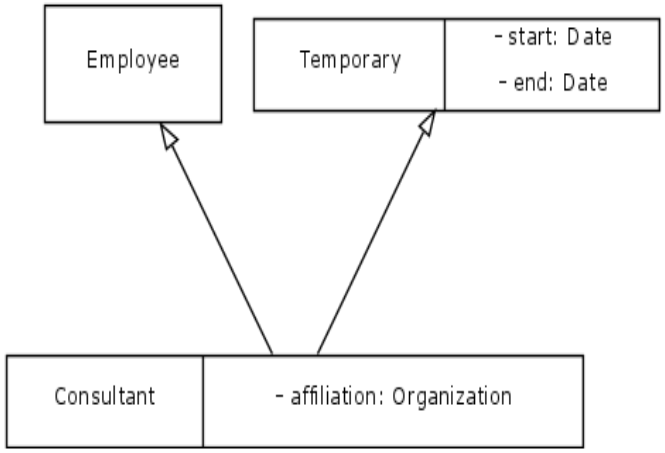
Multiple inheritance

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)

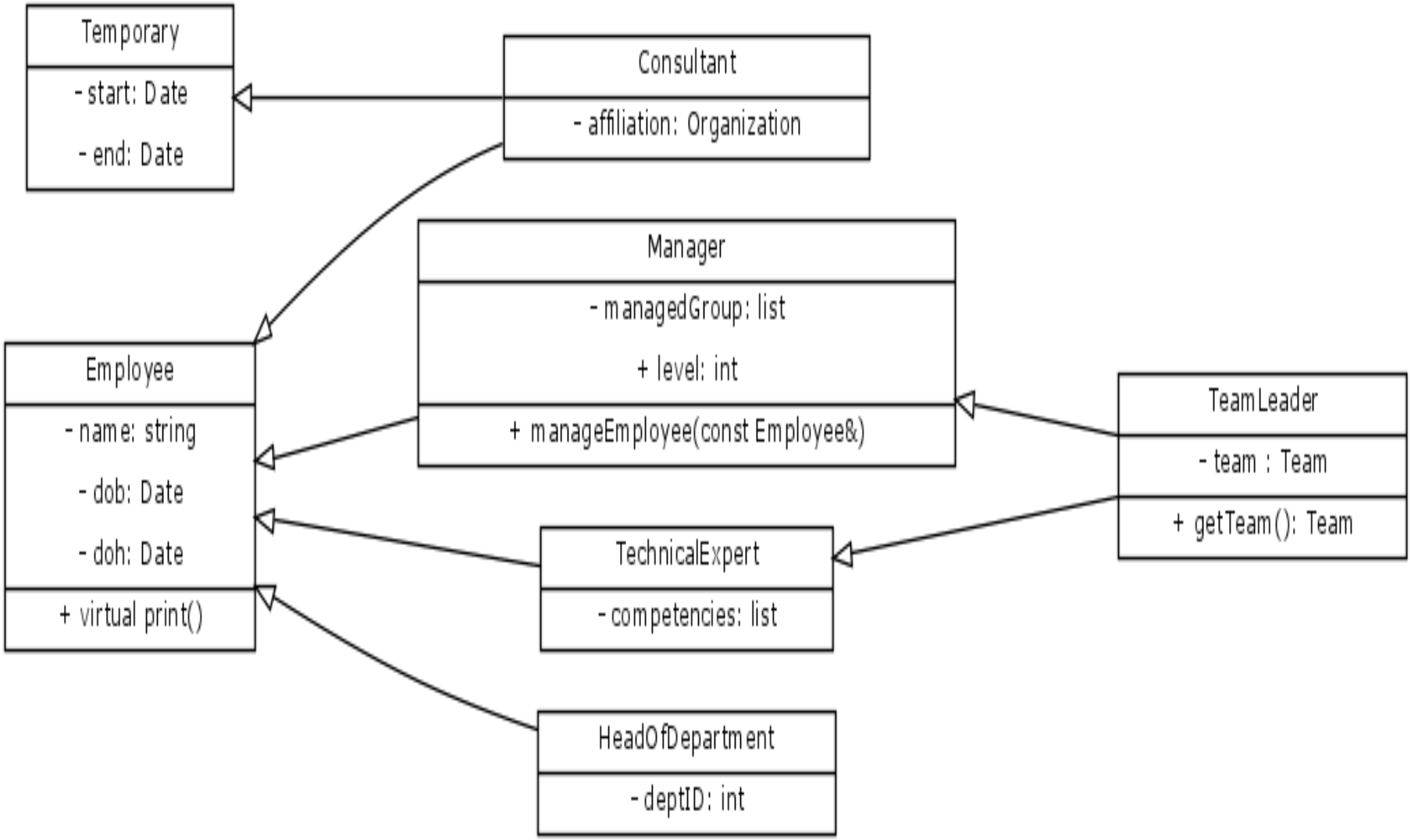
Syntax

```
class DerivedClass : [access_modifier1] Base1,
                     [access_modifier2] Base2, . . .
                     [access_modifierN] BaseN {
     // declarations
};
```

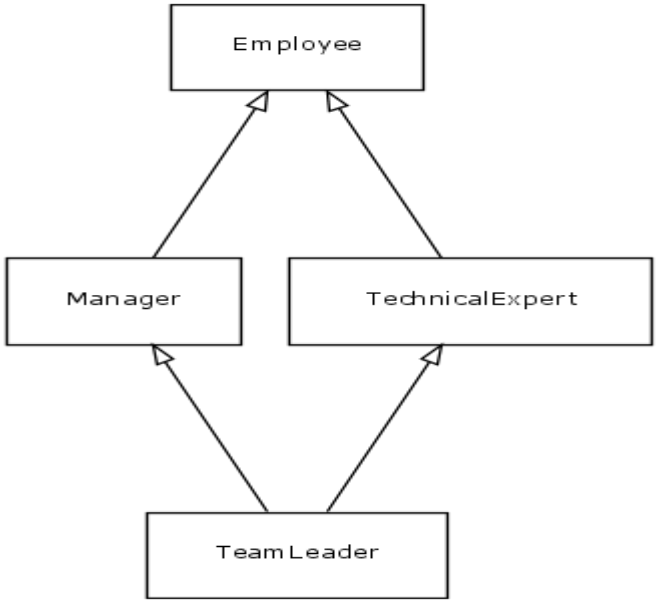



```
class Temporary {
     Date start, end; // period of collaboration
}


// Multiple inheritance – a Consultant is a temporary Employee
// with an external affiliation
class Consultant : protected Temporary, public Employee {
      Organization affiliation;
public:
      Organization getAffiliation();
      void setAffiliation(const Organization& org);
};
```

Employee class diagram - reloaded



Diamond-like multiple inheritance




```
class Employee {
     // declarations
};

class Manager : public Employee {
     // declarations
};


class TechnicalExpert : public Employee {
     // declarations
};


class TeamLeader : public Manager, public TechnicalExpert {
     Team& team;
public:
     Team& getTeam() const;
};
```

Problem: an instance of base class Employee is included twice in an instance of derived class TeamLeader (once from Manager and second from TechnicalExpert), causing:

(1) memory wasting (all data members from Employee are inherited twice)

(2) ambiguities: problems in accessing members from the base class Employee

```
TeamLeader tl;
tl.name = “John”; // refers to Manager::Employee::name or to
                                TechnicalExpert::Employee::name ?!
```

Multiple inheritance and access control

If a member of a base class can be reached through multiple paths in a multiple-inheritance lattice, it is accessible if it is accessible through any path.

```
class Employee {
     // declarations
};

class Manager : public virtual Employee {
     // declarations
};


class TechnicalExpert : public virtual Employee {
     // declarations
};


class TeamLeader : public Manager, private TechnicalExpert {
     Team& team;
public:
     Team& getTeam() const;
};

TeamLeader tl;
tl.name = “John”; // accessible through Manager
```

Virtual base class (I)

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 : [access_modifier] virtual BaseClass {
     // declarations
};
```

Need to explicitly call the virtual base class constructor!

Steps of object initialization:

(1) call virtual base constructor

(2) call constructors of base classes in order of their declaration

(3) initialize derived class members

(4) initialize derived object itself

```
class Employee {
     // declarations
};

class Manager : public virtual Employee {
     // declarations
};


class TechnicalExpert : public virtual Employee {
     // declarations
};


class TeamLeader : public Manager, public TechnicalExpert
     : Employee{...}, Manager{...}, TechnicalManager{...} {
     // declarations
};
```

Virtual base class (II)

Using virtual base, the Employee's members are inherited only once in TeamLeader, thus,

```
TeamLeader tl;
tl.name = "John";
```

unambiguously refers to member name inherited ONCE from Employee.

Note: Other OOP languages (Java) forbids multiple inheritance and replaces it with multiple interfaces implementation + single class inheritance.

Memory representation of class instances for both cases: without and with virtual base classes.

Design considerations

DEFINITION [Interface inheritance] – usage of abstract class inheritance

DEFINITION [Implementation inheritance] – usage of base classes with state and/or defined member functions.

Combinations of the two approaches are possible, we can define and use base classes with both state and pure virtual functions.

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

struct Point {
    double x, y;
}

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

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

Class inheritance vs. object composition

Class inheritance (white-box)	**Object composition (black-box)**
Visibility	Reuse
Static (compile time)	Dynamic (can change at run-time through instantiation)
Easy to understand (and use)	Difficult to understand
Breaks encapsulation principle	Doesn’t break encapsulation principle
Reusing problems	Keeps each class encapsulated and focused on one task
Large class hierarchies; fewer objects	Small class hierarchies; more objects

Delegation pattern

DEFINITION [Delegation] Delegation is handing of a task over another object.

Alternative to inheritance.

Advantage over inheritance: behavior can be changed at run-time

```
class A {
public:
   virtual void foo() {
      printf("An A at work.");
   }
};

class AA : public A {
public:
   virtual void foo() {
      printf("An AA at work.");
   }
};
```

```
class B {
   A* pa;
public:
   B(A* aa) : pa(aa) { }

   void setA(A* aa) { pa = aa; }

   virtual void foo() {
      // delegate the task to object pa
      pa->foo();
   }
};


void main() {
     B b(new A);
     b.foo(); // A behavior
     b.setA(new AA);
     b.foo(); // AA behavior
}
```

Delegation is best used when you want to use another class's behavior **as is**, without changing that behavior. Example: Board, 3DBoard example

Further Reading

1. [Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 12]

2. [Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 20]