Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Classes and polymorphism

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

Hierarchy of concepts

Very often mathematical concepts (but not only) are hierarchical: a triangle and a square are polygons; a P1 and P2 finite elements are finite elements.

It means that we expect to be able to carry out on triangles and squares operations that are in fact common to all polygons. And the same for finite elements.

This type of relationship is expressed in C++ through inheritance (in particular public inheritance) and polymorphism.

Inheritance

Inheritance is basically a mechanism to extend a class, indeed it may be used to build a class from basic components, see CompositionWithVariadicTemplates as example of use of variadic templates with inheritance.

It is also however also the mechanism by which we implement polymorphism: the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior.

Public inheritance

```
class D: public B{....
```

The mechanism of public inheritance is simple:

1. Public and protected members of `B` are accessible by `D`. If they are redefined in `D` you can still access members of `B` using the qualified name (`B::..`).
2. Public members of `B` are public also in `D`.
3. Protected members of `B` are protected in `D` and thus accessible only by `D` and possible classes publicly derived from `D`.
4. Private members of `B` are inaccessible by `D`.
5. Methods defined in `D` hide methods with the same name in `B` (the methods of `B` are still accessible using the qualified name)

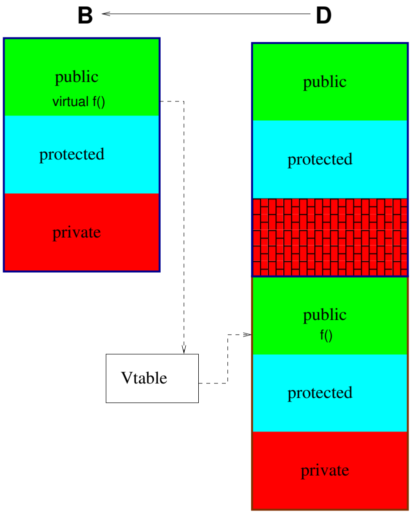
Polymorphism

Public inheritance allow public polymorphism:

1. A pointer and a reference to `D` is automatically converted to a pointer (reference) to `B`.
2. Methods declared virtual in `B` are overridden by methods with the same signature in `D`.
3. If `B* b=new D` is a pointer to the base class converted from a `D*`, calling a virtual method (`b->vmethod()`) will in fact invoke the method defined in `D`.

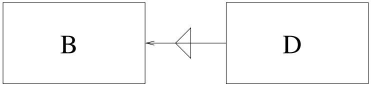
Note: you must have the same return type in overridden virtual methods. With one exception: a method returning a pointer or reference to a base class may be overridden by a method returning a pointer (reference) to a derived class.

Public inheritance and polymorphism



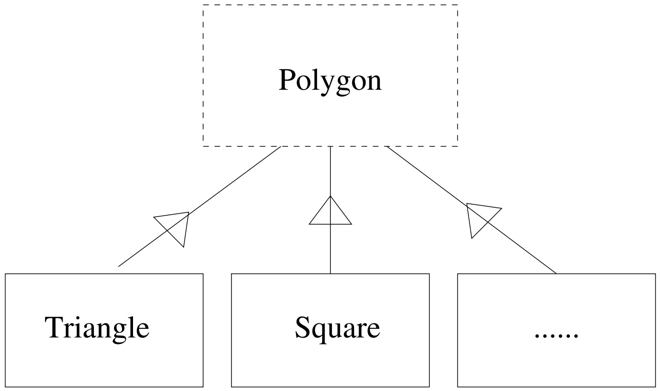
Is-a relationship

Public inheritance polymorphism (simply called polymorphism) must be used only when the relation between base and derived class is an 'is a' relation: the public interface of the derived class is a superset of that of the base class.
That is, I can use on an object of the derived class any method of the public interface of the base.



PROMISE NO LESS, REQUIRE NO MORE (H. Sutter)

An example



Polygon/Polygon.hpp

Overriding

```
void f (Polygon const & p){
...
p.diameter();
...
int main() {
Square s; ...
f(s);
```

I am giving to `f` a `Square` as actual argument. It is possible since the value is passed by reference (conversion rule).
Yet `d.diameter()` in `f()` will now call the method defined in `Square` and not that of `Polygon`, since `diameter()` is a *virtual method* of `Polygon`.

Using pointers

```
void f (Polygon * p){
...
p->diameter();
```

I obtain the same result as before. But if I do

```
void f (Polygon p){
...
p.diameter();
```

here the function accepts only a `Polygon`!

A test

```
class B{
public:
void f1(){cout<<''B::f1''<<endl;}
virtual void f2(){cout<<'' B::f2''<<endl;}
};

class D: public B {
public:
void f1(){cout<<''D::f1''<<endl;}
virtual void f2(){cout<<'' D::f2''<<endl;}
};
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();
b.f2();
d.f1();
d.f2();
dp->f1();
dp->f2();
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();
d.f1();
d.f2();
dp->f1();
dp->f2();
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();
d.f2();
dp->f1();
dp->f2();
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();
dp->f1();
dp->f2();
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1();
dp->f2();
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2();
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1();
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1(); B::f1
br.f2();
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1(); B::f1
br.f2(); D::f2
bp->f1();
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1(); B::f1
br.f2(); D::f2
bp->f1(); B::f1
bp->f2();
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1(); B::f1
br.f2(); D::f2
bp->f1(); B::f1
bp->f2(); D::f2
d.B::f1();
```

What is appearing on the screen?

```
D d; B b; D * dp=new D;
B & br(d); B * bp(&d);
b.f1();   B::f1
b.f2();   B::f2
d.f1();   D::f1
d.f2();   D::f2
dp->f1(); D::f1
dp->f2(); D::f2
br.f1(); B::f1
br.f2(); D::f2
bp->f1(); B::f1
bp->f2(); D::f2
d.B::f1(); B::f1
```

Costruction of a derived class

The constructor of an object a derived class follows this simple rule:

- Variables members inherited from the base class are built using the default contructor of the base class or the base class constructor indicated in the initialization list of the constructor.
- Possible member variable added by the derived class are then constructed, eventually using the constructors indicated in the initialization list, with the usual precedence rule.

Example:

```
Square::Square(Vertices const & v):Polygon(v)
{// possible code ;}
```

As a consequence members of the base class are available to build members of the derived class, if needed.

Derived class destruction

An object of the derived class is destroyed by

- First destroying the variable members defined only in the derived class, in the inverse order of their declaration;
- Then destroying those of the base class, with the usual rule.

Note: If you apply polymorphism the destructor of the base class should be defined `virtual`! This is compulsory if the derived class introduces new variable members.

```
Polygon * p=new Square(v);
delete p; //I need the Square destructor!
```

Things to remember in public inheritance

- Do not change in the derived class the return type and the value of default arguments of a virtual method of the base class;
- Declare the destructor of a base class `virtual`. Possible exception is when the derived classes do not add variable members;
- If you provide a public method in the derived class with the same name but different signature than that in the base class, bring the method of the base class into the scope of the derived class with the `using` statement (but why are you doing it in the first place?, are you participating to the obfuscated C contest?).

```
class Base {
public:
double a(double const &);
double b(int &);
...
};
class Derived: public Base
{public:
using Base::a;
using Base::b;
int a(int const &);
int b(int &);
...};
```

C++11 goodies: the final and override specifications

In the old standard it was not so clear how to specify that a class was not meant to be derived from. Or, to indicate that a method was overriding a virtual method of the base class. One could use some tricks, with the risk of making the program obscure. C++11 introduce two new specifications: final and override

```
struct A{
    virtual void foo() final;
    virtual double foo2(double);
...};
// in a struct inherit. is public by def.
struct B final : A {
    void foo(); // Error: foo cannot be overridden:
                //  it's final in A
...};
struct C : B // Error: B is final
{...};
```

```
struct A{
    virtual void foo();
    void bar();
...};

struct B : A
{
    void foo() const override; // Error:
                //Has a different signature from A::foo
    void foo() override; // OK: base class contains a
                        // virtual function with the same signature
    void bar() override; // Error: B::bar doesn't
                  // override because A::bar is not virtual
};
```

Abstract classes

Sometimes the base class expresses just a concept and it does not make sense to have concrete objects of that type. In other word the base class is meant just to define the common public interface of the hierarchy, but not to implement it (at least not in full).

To this purpose C++ introduce the idea of *abstract class*, which is a class where at least one virtual method is defined as null.

Example of abstract class

```
class Polygon {
public:
explicit Polygon(Vertices v, bool convex=false);
virtual ~Polygon();
bool isConvex() const;
virtual double diameter() const=0;
// ...
protected:
bool isconvex;
};
```

The method declared null must be overridden in a derived class.
Moreover `Polygon p;//ERROR. Polygon is abstract`

dynamic_cast<T>

The command dynamic_cast<T> may be used to check to which derived class a pointer (or reference) to a base class refers to:

```
B* b=new D;
D* dp=dynamic_cast<D *>(b);
```

dp is a pointer to D if D derived publicly from B. Otherwise dp is the null pointer.

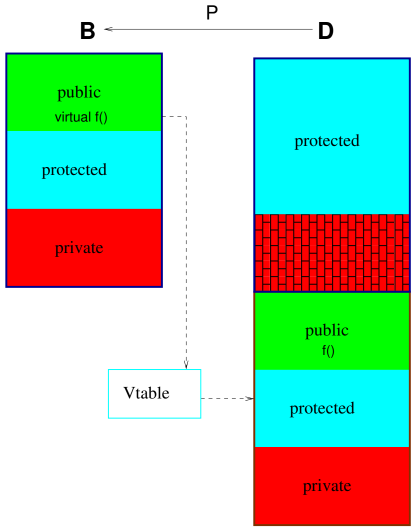
An alternative is to use *run time identification* (RTI) of the *Standard Library* (not covered in the course).

Checking at compile time if a class derived from another may be made with type traits (useful in template programming, we will see later on...)

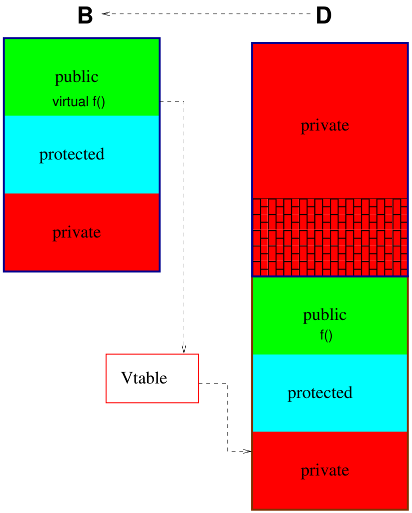
There are two other types of inheritance, protected and private (private inheritance is the default for classes, that's why you need the keyword `public` to indicate the private one, for `struct`s the default is public)

- `class D: protected B`. Public and protected members of `B` are *protected* in `D`. Only methods and *friend*s of `D` and of classes derived from `D` can convert `D*` (or reference) into `B*`.
- `class D: private B`. Public and protected members of `B` are *private* in `D`. Only methods and *friend*s of `D` can convert `D*` into `B*` (it applies to references as well).

Protected inheritance



Private inheritance



Multiple inheritance

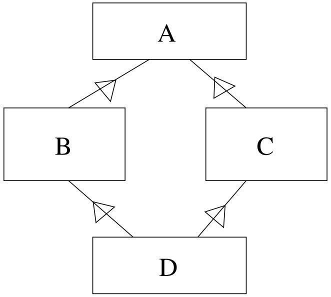
In C++ it is possible to derive from more than one base.
The derivation rules apply to each base class.

Ambiguity in the names may be resolved using the qualified name:

```
class D: public B, protected C{
...
B::pippo(); //I am using the method of B
```

The dreaded diamond

Multiple inheritance can give problems if a class inherits multiply from classes which have a common base. The classical situation is that of the so-called dreaded diamond



In this situation the class `D` has multiple copies of the member data of class `A`, there is ambiguity in the call of virtual methods and pointers and references to `D` are not convertible to pointer to `A` directly!.

Example

```
class A{
public:
virtual double run();
protected:
double my_data};
class B: public A{
public:
virtual double run();
void g(double x)...};
class C: public A{
public:
virtual double run();...};
class D: public B, public C{
public:
virtual double run();
void f();};
void A::f(){...
 double a=g(4.8); // WHICH g()!!!
 a += this->my_data; // WHICH my_data!!}
```

I have two problems here: The compiler does not know which `g()` and which `my_data` I am referring to. The one inherited through B or the one inherited through C??

The solution: virtual inheritance

```
class A{
...};
class B: virtual public A{
...};
class C: virtual public A{
...};
class D: public B, public C{
public:
virtual double run();
void f();};
```

In this way the compiler will copy only one instance of the members of A into C and a pointer (reference) to D is now convertible to a pointer (reference) to A. Polymorphism works!. See the example in DreadedDiamond.

Suggestion

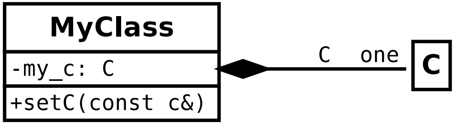
Try to avoid to fall into this situation. First of all virtual inheritance introduces some inefficiencies, but it is not this the most important point here.

Virtual inheritance introduces more complexity in the code. Try to keep your code simple!. However there are situations where virtual inheritance is necessary, for instance when you want to combine classes you have already developed. Yet, in this case try to verify if simple containment is not a better choice.

How classes can collaborate with each other

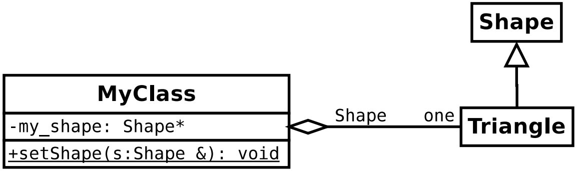
Classes introduce concepts that are often related each other. We have different type of collaboration possible beside the "is-a" relationship.

The simplest collaboration is that of the composition obtained by having an object of another class as member. Typically the object is kept private, and is used within the class. The lifespan of the contained object coincides with that of the container.



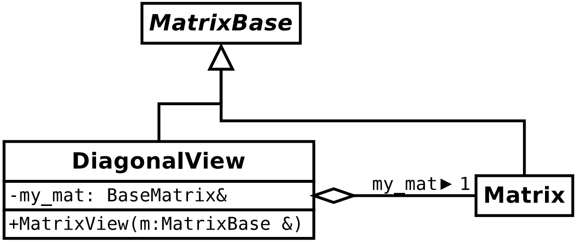
Aggregation

A second type of collaboration is the aggregation. The class stores a pointer or a reference to a polymorphic object. In the case of a reference, it must be initialized in the constructor and cannot be reassigned. The lifespan of the referenced object is independent from that of the containing one, and the former outlive the latter. It the aggregated object is used to define part of the implementation of the class is called a policy.



Views (Proxies)

A view (or proxy) is a particular type of aggregation (usually performed with a reference), whose role is to enable to access the contained element using a different (usually more specialized) interface for particular use. For instance accessing a general matrix ad a diagonal matrix. Is a particular instance of the decorator design pattern.



Implementation details

```
class Matrix{
public:
//! Returns A_ij
double & operator(int i, int j);
...
};

class DiagonalView{
public:
 DiagonalView(Matrix & m):MM(m){}
 double & operator(int i,int j){
 return i==j?MM(i,i):0.0;}
 ...
private:
 Matrix &MM;
};
```

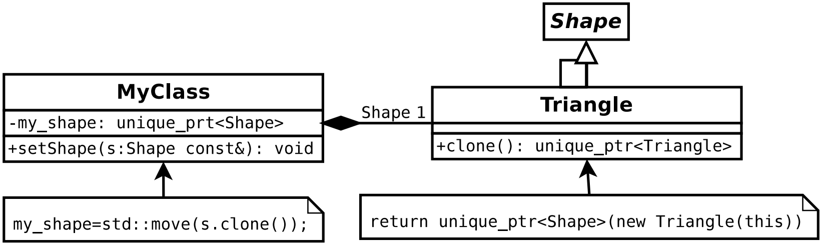
Composition with a polymorphic object

In this case you store a pointer (sometimes a reference) to an object whose lifespan has to coincide with that of the containing class.
We say that the containing class has full ownership of the contained object: it has the responsibility of its creation and destruction. Normally the contained object is created by a object factory (i.e. a function which creates the object) or through the use of the virtual construction (cloning) technique. We will more examples in later lectures.

The control of the lifespan is greatly facilitated by the use of smart pointers (also the subject of another lecture).

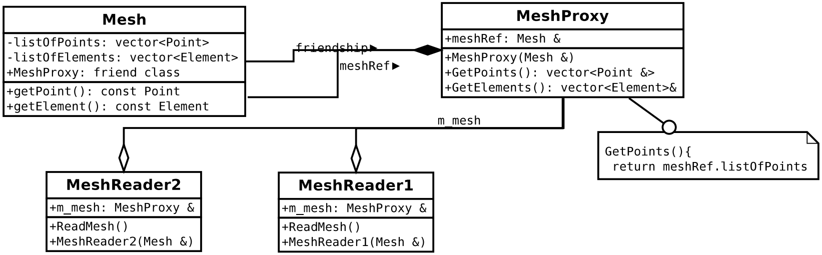
Composition with a polymorphic object

Example of the use of the cloning technique.


Shape
MyClass
-my_shape: unique_prt<Shape>
+setShape(s:Shape const&): void
Shape 1
Triangle
+clone(): unique_ptr<Triangle>
my_shape=std::move(s.clone());
return unique_ptr<Shape>(new Triangle(this))

Another use of a proxy

To create a class that may write on an object which otherwise has only `const` methods (read-only).



Implementation details

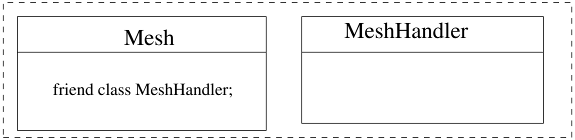
```
class Mesh{
public:
...
// Returns i-th element
Element  operator[](int i) const;
friend class MeshHandler;
private:
Element & operator[](int i);
...
};
```

MeshHandler may modify the class content

```
class MeshHandler{
public:
MeshHandler(Mesh & m):MM(m){}
Mesh & theMesh(){return MM;}
//non-const access
Element & operator[](int i){ return MM[i];}
private:
Mesh & MM;}
```

Friendship

We have seen that class can be declared friend of another class. This way the friend class may access its private member. Remember: membership is a strong relationship: a class declared friend may be considered as part of the implementation.



Delegation

The term delegation is often used to indicate that part of the implementation of a class is made using methods provided by another class.

We often use the term policy if the “delegated object” is polymorphic (or passed as template argument...)

Inheritance is also in a way a delegation since we provide the interface of the base class to the derived one. However public inheritance is a much stronger relation among classes: it implements the “is-a” relationship.

As an alternative we may use protected/private inheritance.

Delegation with protected inheritance

We can use protected inheritance to provide delegation exploiting the `using` statement:

```
class B{
public:
double fun(double);
..}
class D: protected B{
public:
using B::fun;
...
```

Now `fun` defined in `B` becomes part of the public interface of `D`. The following code is correct

```
D myD;   double x=myD.fun(7.0);
```

How do not shoot oneself in the foot?

- Use public inheritance and related polymorphism only to implement an “is-a” relation: promise no less, require no more.
- Prefer composition to express a simple delegation (“is implemented in terms of” as opposed to “is-a”).
- Use protected inheritance to implement delegation only if advantageous.

Composition or protected/private inheritance?

From H. Sutter *Exceptional C++* there are simple rules if you are undecided whether to use composition or private inheritance:

Use private/protected inheritance `B` $\leftarrow$ `D` if

- We need to override virtual methods of `B`
- We need access to protected members of `B` (alternative: use *friend*)
- Class `B` contains only methods (empty class optimization)
- We want a *controlled polymorphism*, applicable only by methods of `D` and classes derived from `D` (you need protected inheritance).

In all other cases, simple composition is better (and this situation is in fact the most common one).