

# Advanced Programming for Scientific Computing (PACS)

## Lecture title: Classes

Luca Formaggia

MOX  
Dipartimento di Matematica  
Politecnico di Milano

A.A. 2023/2024





# Classes and struct

I assume you already know what a class is, we will recall here only the main concepts.

Classes are one of the main features of C++ and are the basic tools for object oriented (OO) programming. I wish to recall, however, that C++ is not an OO language: it's a language that supports functional, object oriented and generic programming, and you normally obtain the best by combining those features.

Classes are meant to enucleate concepts<sup>1</sup> and functionalities, and classes may collaborate in different ways for implementing more complex concepts.

---

<sup>1</sup>I use the English meaning of the term concept, not C++ concepts     

# An example

The concept of a mesh, i.e. a partition of a  $nD$  domain into simple polygons is a complex concept made of

- ▶ Geometric elements: a single type (e.g. triangle) or of a **family** of types (polygons) which themselves need the concept of Points, and of elements at the boundary.
- ▶ Containers for geometric elements and their relation;
- ▶ Tools to compute quantity of interest: area/volume, barycenter, etc. etc.

Identifying the basic components and enucleating them in classes allows a better re-use, documentation and **testing**.

# Classes in C++

A **class** object is defined by using either the keyword **class** or **struct**. They are "almost" synonyms: the only difference is the default access rule: **public** for **struct** and **private** for **class**.

It is a common habit (but not compulsory) to limit the use of **struct** to define just collection of data publicly accessible, more precisely **aggregates**, while **class** is used to indicate a more complex type where data can be not publicly accessible.

With a class you are effectively introducing a new **type** in your code.

From now on I use the term *class* and *class-type object* to indicate a type, and the corresponding object, introduced by a **class** or **struct** declaration.

# What can a class can contain?

Class members can be:

- ▶ **Types**: either type alias, defined with the **using** construct, or **nested classes**;
- ▶ (non-static) **variables** (or data members). They form the **state** of an object of that class.
- ▶ **static variables** (or static data members). They can be addressed also with the **scope resolution operator** (`::`). Introduced by the keyword **static**. They form the **state** common to all objects of that class.
- ▶ (non-static) **methods** (or function member). They can access data members (static and non static).
- ▶ **static methods**. They can access only static members of the **class**. Introduced by the keyword **static**.
- ▶ **static constant expressions**. Only static (well it would not make sense otherwise)

# What a class represent: the single responsibility principle

**The single responsibility principle:** Every class should be responsible of a single and well identifiable part of the functionalities of a computer code.

Classes expose a well define public interface which should not change, but may be extended (open-closed principle). Only this way classes can be easily reused and collaborate with each other to make up your program or library.

Often classes are designed with a bottom-up approach: classes representing simple well identified components, are developed and **tested separately**, and then this basic components collaborate to form classes representing more complex functionalities.

# In-class definitions and initialization

Methods can be defined “in-class”. Non static data members may be initialised in-class and also static data by declaring them **inline**;

```
class MyClass
{
    ...
    public:
        // in-class definition
        double get_x() const {return x;}
        double a{5.0}; // in class initialization
        inline static float c=6.0f; // in class initialization
};
```

In class method definition and data initialization are part of the class declaration so they are in a header file!

## Out of-class definitions and initialization

Alternatively methods and static member variables may be defined out-of class in a source file. This is normally done for long methods.

"Myclass.hpp"

```
class MyClass
{
public:
    double get_x() const; //only declared
    double a{5.0}; //
    static float c;//only declared
};
```

"Myclass.cpp"

```
#include "Myclass.hpp"
double MyClass::get_x()const {return x;}
MyClass::c=6.0f;
```



# Member access

Non-static members are accessed via the **member access operator** (e.g. `a.x`), if you have a pointer to an object of the class you use the other form of member access operator: `pa->x`.

Static members may be accessed also with the scope resolution operator (e.g. `Myclass::x`), it means that we do not necessarily need to have an object of the class to access them.

Non-static member functions can access member variables of the class or just using their name, or with the **this** pointer: **this**->x.

Static member functions can access only static member variables of the class, using their name or the `::` operator: `MyClass::s`.

# The **this** pointer

Any non-static function members of a class has access to a special pointer that is the pointer to the object that is calling that member function:

```
struct Foo{  
  double fun(); // pure declaration  
  double fun()const; // pure declaration  
  double gun(); // another method  
  double y;  
};  
double Foo::fun(){  
  this→y= 0; // this is a Foo*  
  ...  
}  
double Foo::fun() const{  
  auto x = this→y; // this is a Foo const *  
  ...  
}
```

However C++ spares you the need to indicate **this** explicitly when accessing a member.

## const methods of a class

A (non-static) member function that does not change the state of the class should be declared **const**.

Only const methods are available in a const object of the class.

The **const** specifier is part of the member function signature and participates to overloading. If the same method is present with its const and non-const version, the non-const version is used on non-const objects.

```
Foo foo;  
Const Foo cfoo;  
foo.fun(); // double fun() is called  
cfoo.fun(); //double fun() const is called  
cfoo.gun(); // SYNTAX ERROR! gun() is not const!
```

I repeat: always declare const methods that do not change the state of the class.

# The **this** pointer

The **this** pointer is the equivalent of `self` in Java or Python. And indeed if we look at the symbols generated from `Foo::foo()` and `Foo::foo()` **const** we find

```
Foo::foo(Foo * this)
Foo::foo(const Foo * this)
```

The language automatically adds an extra parameter that is set as the address of the calling object automatically. C++ does that for you, no need to bother.

The **this** pointer is the mechanism with which non static member functions can access other members. You normally don't have to indicate it (which is a nice feature of C++, in python you have to write `self.` to address a member), but we will see in another lesson that when dealing with templates it may be safer to indicate **this** explicitly.

## mutable members

If a class object is declared **const**, you can **only use const methods** and not change its state. In some particular cases, it may be necessary to have some data members modifiable **even on constant objects**. You may use **mutable**.

```
class Foo{
public:
    Foo(double a):a{a}{}
    double getA() const {
        return a;
        done=true;      }
private:
    double a=0.;
    mutable bool done=false;
}

....
const foo s{3.} // a=3.0, done=false
auto b=s.getA(); // done=true
```

# Access rules

Class members can be given different access rules:

- ▶ **public:** For everybody!. All can access the member. We say that the member is part of the public interface of the class.
- ▶ **private:** Only for me!. Only member functions of the class can access a private member. No one else!
- ▶ **protected:** For me and my family!. The member is accessible only by methods of the class and of publicly (or protected) derived classes.

# An example of simple class

```
class Foo
{
    public:
        using Real = double; //a type alias
        Real a=10.0; // in-class initialization
        int j; // Another non-static member variable
        // To initialize a static member variable I need inline (C++17)
        inline static Real c=10.89;
        static constexpr Real pi=3.1415;
        double getV()const {return v;} // a getter
        void setV(const Real & a){v=a;} //a setter
        Real & setV{return v;} // A more C++ style setter!
        static void setX(Real const & a){x= a;} // sets a static variable
    protected:
        std::vector<double> z_; // a protected member
    private:
        inline static Real x; // A static member variable
        Real v;
};
```

# An example of simple class usage

```
// Setting a static variable  
// I do not need an object since it is static  
Foo::setX(10.0);  
// Object default constructed;  
Foo p;  
// I can get the static variable  
// also with member access operator  
auto constexpr x=p.pi;  
// But I can do also:  
// auto constexpr x=Foo::pi;  
p.setV(10.0); // setting value of p.v  
p.setV()=10.0;// using the more C++ style setter  
using Real = Foo::Real; // extracting type  
Real y{p.getV()};// Initializing y with p.v  
auto pf=std::make_unique<Foo>(); // a unique_ptr<Foo>  
pf->setV(90.);// Accessing through pointers
```



## Another example of static variables and functions

```
class TriaElement{
public:
...
// a constexpr member must be static!
static constexpr int numnodes=3;
// Here a static constexpr function
static constexpr int NNod(){return 3;}
};
...
//I can access the static variable with no object
std::array<int, TriaElement::numnodes> nodeID;
//But I can use also the constexpr method instead
std::array<int, TriaElement::NNod()>;
```

## Synthesized (or automatic) methods

When you create a class T you have the following fundamental methods automatically defined:

Default constructor	T()
Copy-constructor	T(T <b>const</b> &)
Move-constructor	T(T&&)
Copy-assignment oper.	T & <b>operator</b> =(T <b>const</b> &)
Move-assignment oper.	T & <b>operator</b> =(T&&)
Destructor	~T()

The terms to indicate them are synthesized, but also automatic or implicit.

They can be overloaded by user definitions, or even deleted (a part the destructor, which cannot be deleted)

# Implicit (synthetic) methods

```
class Foo{
    public:
        Foo(int i):i{i}{}; // constructor taking an int
        Foo()=default; // but I want also the implicit default one
        // This class has no copy-assignment operator
        Foo& operator=(const Foo &)=delete;
        // And consequently neither move-assignment
    private:
        int i=0; // I give a default value
};

...

Foo a; // calls synth. default const
Foo b(3) // calls Foo(int)
a=b; // SYNTAX ERROR: no copy-assign.
```

# What do the synthesized methods do?

The synthesized default constructor constructs all non-static variable members using their default constructor or copy-constructor for variables initialized in-class) **in the order they are declared in the class.**

The other methods (a part the destructor) operate similarly, just by changing the verb: a copy-constructor copy-construct the non-static member variables, assignment operators assign them (by copying or by moving), always **in the order they are declared in the class.**

The destructor calls the destructor of all non-static member, **in the opposite order with respect their declaration in the class.**

Static variable members are constructed "at compile time".

# Rules for automatic methods

The synthesized methods are defined automatically only if the user do not declare them explicitly.

Moreover:

- ▶ A default constructor is automatically generated only if the user does not declare any other constructor;
- ▶ Move operations are generated automatically only if: a) No copy operations are user declared; b) No move operations are user declared; c) No destructor is declared;

If you want to "resurrect" synthesized methods when they are not generated automatically, use the keyword **default**. If you do not want them to be generated at all use the keyword **delete**.

# The rules for the generation of implicit methods

compiler implicitly declares							
user declares		default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
	Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
	Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
	default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
	destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
	copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
	copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
	move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
	move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

**Not declared:** not declared by the compiler, but the user may recall it using the `default` keyword. Deleted: not provided by the compiler, but the user may still define one. See [here](#) for details.

## (User defined) default constructor

A default constructor is **any constructor that may take no arguments**. A class with a default constructor (user defined or automatic) is called **default constructible**.

```
class FirstClass{
public:
    // declaration of a user defined default const
    FirstClass();
    ...};

class SecondClass{
public:
    // Also this is a def. constr.!!
    SecondClass(int i=5):my_i(i){} // defined in-class
private:
    int yy_i;
};
```

Only if a class is default constructible you can do

```
MyClass m; // or
MyClass b{};
```

# Constructors in general

The in-class definition of a constructor has the form

```
ClassName(<Param>):<Init. List>{<Body>}
```

Here <...> indicates optional parts. An out-of-class definition:

```
ClassName::ClassName(<Param>):<Init. List>{<Body>}
```

The initialization list is used to **initialize** non-static variable members. Members not initialized in the initialization list are initialized by their **default constructor** or copy-constructor if initialized in-class.. **Members are initialized in the order declared in the class, not that in the initialization list.** The body may contain other actions, different from initialization. In the body scope, all members are available. **Initialize members in the initialization list, do not assign them in the body of the constructor, it is more efficient!** The body can be empty (but you need the { }).



## Example

```
class Foo
{
    public:
        Foo()=default; // default constr. is the synthesized one
        Foo(double x, std::vector<double> const & v): x_{x}, v_{v}{};
    private:
        double x_;
        std::vector<double> y_
};

...
Foo foo; // default constructed (x_ undefined, v_ empty)
Foo foo2{3.4,{4,,6.,-9.89}}; //x_=3.4, v_ size=3
```

# Delegating constructors (part I)

You have the possibility of calling another constructor within a constructor. Sometimes it is very useful.

```
class Foo{  
public :  
Foo(int i=0); // A constructor taking a int  
Foo(int i, double x):Foo(i)  
    { ... /*do something*/ }  
}
```

I use the constructor Foo(**int**) inside the initialization list of the other constructor.

# Destructor

The **destructor** is a special method of the class with name `~ClassName`. It has no return type nor arguments. Normally, it is required to write a destructor only if the class handles memory dynamically through pointers:

```
MyMat0::~~MyMat0() { delete [] data; }
```

**An important note:** If the class is used as a base class **the destructor should be declared virtual!**.

**Another Note:** Use containers or unique pointers to store dynamic data owned by the class. You avoid the fuss of writing a destructor (and possible bug hunting headaches).

# Copy constructor

The copy constructor is called when we initialize an object by copying from another object of the same type: `MyMat0 a(b)`, or `MyMat0 a=b` calls the copy constructor of `MyMat0`.

The synthesized one just copies the members (using their corresponding copy-constructor) one by one, **in the order of declaration**.

You normally need to write your own copy constructor if you store a pointer to dynamically allocated data and want a **deep copy (i.e. to copy the pointed data and not the pointer itself)**. But again, if you use a standard container you are spared the problem.

A class with an accessible copy constructor is called **copy-constructable**.

# When do you need to write your own copy constructor?

Typically, it happens when in your class you handle resources through pointers and you want to deep-copy the resource. For instance, let assume that `MyMat0` is a matrix that stores the data via a unique pointer `unique_ptr<double[]> M_data`. We can do

```
MyMat0(MyMat0 const & m): nr(m.nr), nc(m.nc)
{
    M_data=make_unique<double[]>(m.nr*m.nc);
    for (unsigned i=0;i<m.nr*m.nc;++i)M_data[i]=m.M_data[i];
}
```

If you do not define the copy constructor, the synthesized one performs a **shallow copy**: copies the pointer not the resource, probably not what you want!

**Note:** If store the data in a standard vector you are saved the fuss! You delegate the copying to the vector, which performs a deep-copy (one responsibility principle).

# Copy-assignment

Again, you normally need to write your own copy-assignment if you aggregate resources through pointers. It is important to ensure correct behavior when you do  $a=a$ , which should be a **no-operation**. And remember to **return a reference to the object**, in order to be able to do  $a=b=c$ .

```
MyMat0 & operator=(MyMat0 const & m){  
    if (this!=& m); // handles a=a  
    {  
        M_data.reset(new double[m.nr*m.nc]); // release old data  
        for (unsigned i=0;i<m.nr*m.nc;++i)M_data[i]=m.M_data[i];  
        ...  
    }  
    return *this;  
}
```

A class that has a copy-assignment operator is **copy-assignable**, and you can do  $a=b$  on objects of the class.

# Move-constructor and move-assignment

We postpone the explanation to the lecture on move semantic.

I only anticipate here that you need to define your own move operators only if your class is potentially handling large data dynamically, and you want to define how the data can be "moved" between objects without unnecessary copies. If you use standard containers to store your data the synthesized operators are fine, since containers know how to move the elements saving memory:

```
std::vector<double> v; // a vector of 3 doubles  
v.fill(1000,3.0); // filled with 1000 values  
std::vector<double> b=std::move(v); // move constructor is called  
// b.size()=1000 while v is empty: v.capacity()=0!
```

## A general rule

If there is no special reason to do otherwise, make sure your class is default constructible, copy constructible and copy assignable (and possibly also move-constructable and move-assignable), using the implicit or user declared methods.

If the implicit methods are good for you, there is no need to define your own. If you want to "resurrect" a implicit method that has been hidden because of the rules previously described, use the **default** keyword.

If, for some reason, you want to make sure not to have some of them, use **delete**.

```
struct pippo
{
    pippo()=default; // use the synthetised default constructor
    pippo(const pippo &)=delete; // No copy allowed!
    ...};
```



# The rule of three/five/zero

- ▶ If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three. And if move semantics is desirable, you probably have to declare all five special member functions
- ▶ Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators: **if you can avoid defining them, do it!.**

When a base class is intended for polymorphic use, its destructor may have to be declared public and virtual. **This blocks synthesised move operators** and so, if you need them, **you need to define them or declared them as defaulted.**

## Explicit constructors and implicit conversions

Any constructor that may take a single argument (including the possible effect of defaulted parameters) defines an **implicit conversion** unless declared **explicit**. Sometimes it's what you want, sometimes it's not...in the latter case use the **explicit** specifier.

```
struct Foo
```

```
{  
    Foo(int);
```

```
...
```

```
struct Foo2
```

```
{  
    explicit Foo2(int);
```

```
...
```

```
Foo a=10; // I create an object of Foo type
```

```
a=9;// OK int→Foo conversion
```

```
Foo2 b{10};// An object of Foo2 type
```

```
Foo2 c=20;//Error, no implicit conversion
```

```
Foo x= a+10;// Ok int→Foo
```

```
Foo2 z=b+10;// error int→Foo2 non possible
```

# Casting operator

You want to convert a Foo to an int

```
struct Foo
{
    operator int() { return i; }
    int i;
};

struct Foo2
{
    explicit operator int() { return i; }
    int i;
};

..
Foo a{3}; // Foo is an aggregate
int b = a; // OK Foo → int
Foo2 c{9}; // OK
int z = c; // ERROR!
int z = static_cast<int>(c); // OK explicit cast
```

Use with care!

# Forward declaration

A **Forward declaration** may be used just to tell the compiler that a class type exists.

```
class MyClass; // forward declaration
```

```
..
```

```
double f(const MyClass&); // OK!
```

```
...
```

```
MyClass m; //NO! I need a full declaration!
```

```
auto x=MyClass::staticFun(); // NO! full declaration necessary!
```

If you need just to reassure the compiler that a certain class exists you can use a forward declaration. A full declaration is necessary if I need to create an object or call methods of the class (even static ones).

Forward declarations are necessary to solve some catch22 situations.

## A Catch22 situation

```
class B; // I need it!
class A
{
    public:
        // A has a method that uses a B
        void fun (const B& b);

}
class B
{
    private:
        A M_a; // B stores an A
}
```

Without the forward declaration I'd be stuck!

# The inline directive

**inline** may be applied also to methods of a class.

```
class foo{
  public:
    // Declaration of a inlined method
    inline double method1(int);
    // In-class definitions implies inline!
    double & getValue(i){return my_v[i];}
    ...
};
// Definition (inline not needed here)
double foo::method1(int i){...}
```

If the method is defined in-class is automatically inline (but is not an error if you add the **inline** keyword).

Definitions of inlined member functions should be given in the header file.

# Aggregates

An **aggregate** is a particularly simple class type:

- ▶ Only public non-static data members;
- ▶ No user declared constructor;
- ▶ No virtual member functions;
- ▶ Inheritance allowed but only public and from a base class that is an aggregate.

It is customary to use **struct** to define an aggregate. Note that `std::array`, `std::pair`, `std::tuple` and C-style fixed size array like `double v[10]` **are all aggregates**.

# Aggregates features

Aggregates have some nice features

```
struct NewtonOptions{  
double tolerance=1e-8;  
unsigned int maxIter=100;  
};
```

- **aggregate initialization:**

```
NewtonOptions aa{.001,300};  
NewtonOptions bb={5e-4,60};
```

- **structured binding:**

```
auto [tol,maxIt]=aa; // create a copy  
auto & [tol2,maxIt2]=bb // creates a reference
```



# How classes can collaborate with each other

Classes introduce functionalities that are normally related each other. Different type of "collaboration" among classes are possible.

Let's start with the strongest way to connect classes: creating a hierarchy with **inheritance** and polymorphism.

# 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, normal and binomial **are** distributions.

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 the other examples.

This type of relationship ("is-a") is expressed in C++ through public **inheritance** and **polymorphism**.

However inheritance and polymorphism are related but independent concepts: you need inheritance to apply polymorphism, but you may use inheritance without polymorphism.

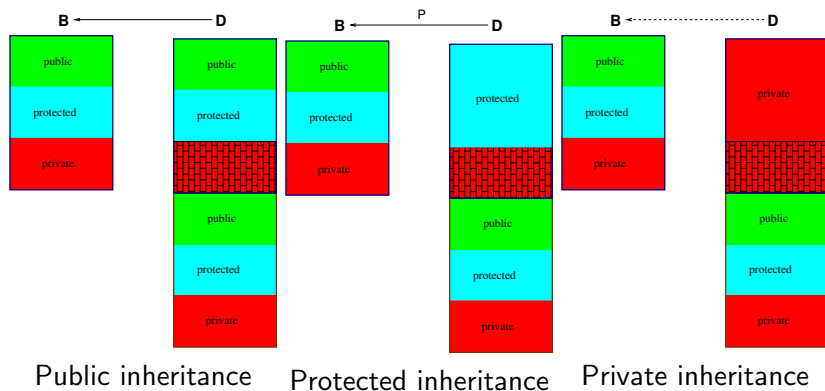
Inheritance is a indeed also possible way to implement more complex concepts starting from simpler ones (composition via public inheritance).

# Inheritance

Inheritance can be public, protected, private. The most commonly used is the public one, but let's explain the different choices.

- ▶ Public: Get everything from my parent, **apart its private stuff**, maintaining its privacy levels;
- ▶ Protected: Get from my parent its public and protected members, but make all them protected: only I and my siblings will be able to use them.
- ▶ Private: Get the public and protected members from my parent, but make them private: only I can use them!.

# A graphical view



# Public inheritance: details

```
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, methods with the same name in B are hidden, but you can still access them 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.

# Construction of a derived class

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

- ▶ First, variables members inherited from the base class are constructed using the default constructor or the rule specified in the constructor of the derived class;
- ▶ Possible member variable added by the derived class are then constructed, following the usual rule.

As a consequence, **members of the base class are available for building members of the derived class**

# Derived class destruction

An object of the derived class is destroyed by

- ▶ First destroying the member variables defined by the derived class, in the inverse order of their declaration;
- ▶ Then destroying those of the base class, with the usual rule.

## Delegating constructor II

In the constructor of a derived class I can call the constructor of the base class. Useful if I need to pass arguments. Otherwise, the default constructor of the base class is used (in which case the base class must be default constructible)

```
class B
{
public:
    B(double x){...};
    ...
};
class D : public B
{
    D(int i, double x):B(x) My_i{i}{}
private:
    int My_i;
};
```



# Inheriting constructors

Constructors are not inherited (but can be recalled with **using**)

```
class B
{
public:
B(double x):x_{x}{...};
...
};
class D : public B
{
using B::B; //Inherits B constructors
private:
int My_i=10;
};
```

Now, `D d{12.0}` calls the `B::B(double)` constructor: `d.x` is set to 12.0 and `d.My_i` takes the default value 10.

# Multiple inheritance

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

Possible ambiguity in the names may be resolved using the qualified name:

```
class D: public B, public C
{
public:
void fun(){
// if both B and C define foo() here I
// resolve the ambiguity using the method of B
auto x=B::foo();

    ...
}

...
}
```

# Composition by inheritance

Inheritance is basically a mechanism to extend a class, indeed multiple inheritance may be used to build a class from basic components

```
class FemElement: public RefShape,  
                  public PolSpace,  
                  public Quadrule  
{...};
```

A FemElement is formed by a reference shape (RefShape), a polynomial space (PolSpace), a quadrature rule (Quadrule).

This type of design, called **composition by inheritance** requires some care, but is sometimes very useful. You normally have a constructor that receives the different components and call the copy-constructor of the corresponding base class (delegating construction):

# Construction of the composed object

```
class FemElement: public RefShape,  
public PolSpace,  
public Quadrule  
{  
  FemElement(const RefShape& r, const Polspace & p,  
              const Quadrule & q ):  
    shape_{r},space_{p},rule_{q}{}{};  
  ....  
}
```

# Construction of the composed object (move friendly)

An anticipation to move semantic. Assuming that for PolSpace "moving" is more efficient than copying:

```
class FemElement: public RefShape,  
public PolSpace,  
public Quadrule  
{  
  template <class R>  
  FemElement(R&& r, const Polspace & p,  
              const Quadrule & q ):  
    shape_{std::forward<R>{r}}, space_{p}, rule_{q}{}{};  
  ....  
}
```

This also show that constructors (as well as assignment operators) can be a template!.

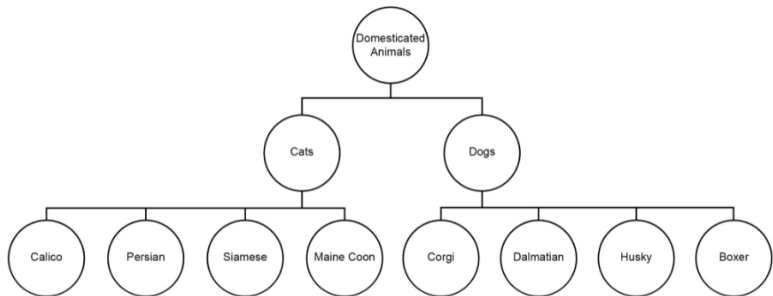
## Another example (for nerds)

(anticipation of the lecture on templates)

Using variadic templates it is possible to make composition by inheritance very elegant, see [CompositionWithVariadicTemplates](#) for an example of use of variadic templates with inheritance.

# Polymorphism

Public inheritance is also however also the mechanism by which we implement **polymorphism**: the ability of objects belonging to different class of a hierarchy to operate each one according to an appropriate type-specific behavior.



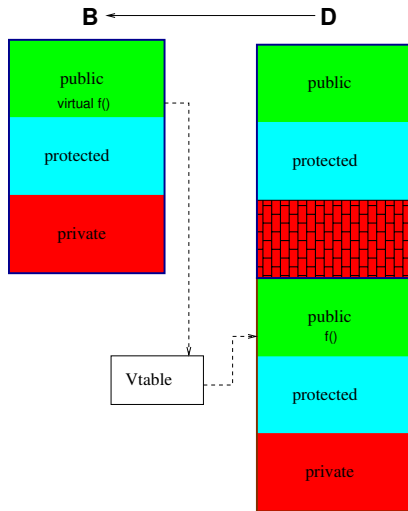
# The mechanism of (public) polymorphism

1. A pointer or a reference to D is implicitly converted to a pointer (reference) to B (upcasting). A pointer or a reference to B can be explicitly converted to a pointer (reference) to D if  $B \leq D$  (downcasting), using `static_cast` (statically) or `dynamic_cast` (dynamically).
2. Methods declared **virtual** in B are **overridden** by methods with the same **signature** in D.
3. If  $B \& b = d$  is a reference to the base class bound to an object of the derived class, calling a virtual method (`b.vmethod()`) will invoke **the method defined in D** It applies also to pointers:  
`B* b=&d; b->vmethod()`.

Overridden virtual methods should have same return type, with one exception: a method returning a pointer (reference) to a base class may be overridden by a method returning a pointer (reference) to a derived class.

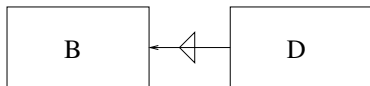


# Public inheritance and polymorphism



# Is-a relationship

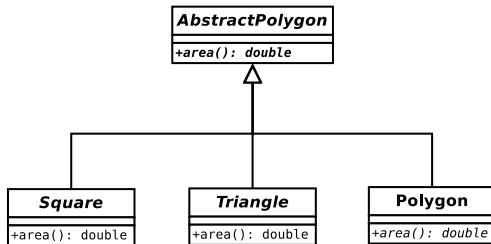
Public inheritance polymorphism (simply called polymorphism) should be used only when the relation between base and derived class is an **is-a** relation: the public interface of the derived class is a superset of that of the base class.



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

It means that one should be able to use safely from an object of the derived class **any member** of the public interface of the base, and this implies that the base class must define the common public interface **of all members of the hierarchy**.

## An example



See the example in [Polygon/Polygon.hpp](#)

# Overriding

```
void f (AbstractPolygon const & p){  
    ...  
    auto a=p.area();  
    ...  
}  
int main(){  
    Square s;  
    ...  
    f(s);  
}
```

I am giving to `f()` a `Square` as argument, passed **by reference**. It is possible since we have the implicit conversion

`Square& → AbstractPolygon&`.

Yet, `p.area()` in `f()` will now call **the method defined in `Square`** and not that of `AbstractPolygon`, since `area()` is a *virtual method* of `AbstractPolygon`.

# Polymorphisms is applied with pointers or references

This is an error:

```
void f (AbstractPolygon p){  
    ...  
    p.area();  
    ...  
}  
int main(){  
    Square s;  
    ...  
    f(s);  
}
```

Two compilation errors: a Square is not convertible to an AbstractPolygon.

Moreover, AbstractPolygon is abstract (see later) so I get a compiler error also in the definition of f().

# Virtual destructor

If you apply polymorphism the destructor of the base class should be defined **virtual**! This is compulsory if the derived class introduces new member variable.

The reason:

```
{  
//create a pointer to a polygon  
std::unique_ptr<AbstractPolygon> p=std::make_unique<Square>();  
}// exiting the scope, resource is deleted
```

When destroying the resources held by p, I need the Square destructor! If I forgot **virtual** that of AbstractPolygon is called instead, and if Square has added new data members we have a **memory leak**

**Note:** The compiler warning `-Wnon-virtual-dtor` issues a warning if you have forgotten a virtual destructor. Most IDE will also warn you if virtual is missing.

# Abstract classes

Sometimes the base class expresses just an abstract 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 AbstractPolygon{
    public:
        explicit AbstractPolygon(Vertices v, bool convex=false);
        virtual AbstractPolygon();
        bool isConvex() const;
        virtual double area() const=0; //null method
        // ...
    protected:
        bool isconvex;
};
```

The method declared null **must** be overridden in a derived class.  
Otherwise also the derived class is abstract.



# The final and override specifications

We have two new specifiers to help avoiding errors: final and override.

- ▶ final for a method means that that method cannot be overridden;
- ▶ final for a class means that you cannot inherit from that class;
- ▶ override tells that a method is overriding one of the base class.

Unfortunately, override is not compulsory (because of backward compatibility), but **I warmly suggest you to use it!**.

**Note:** The option `-Wsuggest-override` tells the compiler to warn you if an override looks missing.

## Examples of final and override specifiers

```
struct A{  
    virtual void foo() final;  
    virtual double foo2(double);  
    ...};  
// in a struct inheritance 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  
};
```

The override specification when overriding virtual member functions makes your code safer. USE IT.

# Checking which class I have

There are two ways to check whether a class is derived from another one:

- ▶ **At compile-time** (statically). We can test at compile time whether a class is base class of another class. Useful in the context of generic (template) programming.
- ▶ **At run time** (dynamically). More expensive computationally, but it allows to test pointers and references to objects of the base class.

# Static testings

Require type-traits (you need `<type_traits>` header.

```
template<class T> void fun(const T & v)
{
    if constexpr(std::is_base_of_v<AbstractPolygon,T>)
    {
        // do something if T derives from Polygon
    } else
    {
        // do something else
    }
}
```

`std::is_base_of_v<B,D>` (or `std::is_base_of_v<B,D>::value`) returns **true** if and only if D derives from B. **if constexpr** is the compile time if: only the code relative to the true branch is compiled.

## Dynamic downcasting: `dynamic_cast<T>`

`dynamic_cast<D*>(B*)` converts a `B*` to `D*` (if `D` derives publicly from `B`). If the condition fails it returns the null pointer, otherwise the pointer to the derived class. It may be used to check to which derived class a pointer to a base class refers to. It works also with references, but if the condition is not satisfied it throws an exception.

```
double fun(AbstractPolygon const & p)
{
    if (dynamic_cast<Square const *>(&p)!=nullptr)
    {
        // It is a square
    } else{
        // It is not a square
    }
}
```

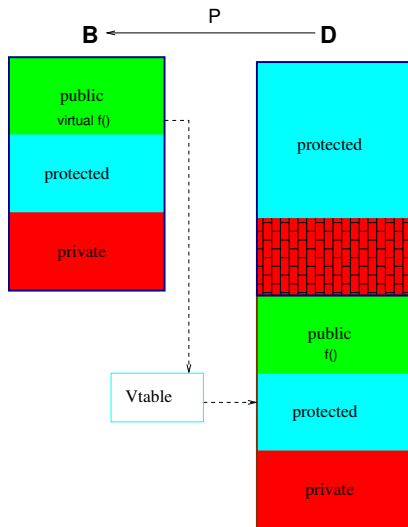
(here I cannot use `if constexpr`)

# Protected and private polymorphism

They use the 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 structs the default is `public`)

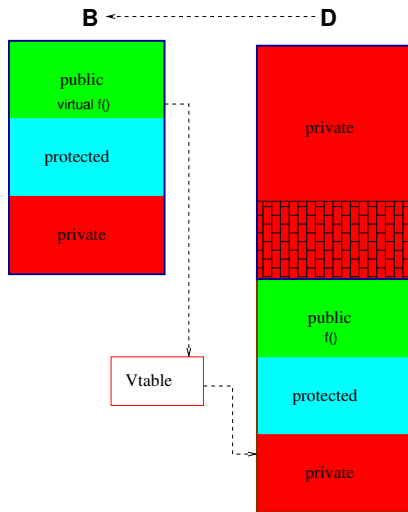
- ▶ `class D: protected B`. Public and protected members of B are *protected* in D. Only methods and *friends* of D and of classes derived from D can convert `D*` (or reference) into `B*`.
- ▶ `class D: private B`. Public and protected members of B are *private* in D. Only methods and *friends* of D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

# Protected inheritance





# Private inheritance



# Why protected/private inheritance?

The use is very special. Typically you use protected polymorphism if you want to use polymorphism, but limiting its availability to methods of the derived classes.

The object is not polymorphic for the “general public”, but only within the class of the hierarchy.

Less common is the use of private polymorphism.

Remember that protected/private inheritance does not implement an “is-as” relationship.

# An example of selective inheritance

I want only part of the public interface of the base class to be exposed to the general public:

```
class Base{
    public:
        double fun(int i);
    ... //other suff
};
class Derived: private Base
{
    public:
        using Base::fun; // fun() is made available
        // other stuff
};
```

Now an object of type Derived may call fun() defined in Base. This is another use of the **using** directive!

# Collaboration among classes

We have seen inheritance, but inheritance is in fact something more than "collaboration", it is more an embedding.

We look now at the most common ways classes collaborate with each-other.

# Dependency

We say that class A depends on class B if A uses B. So this is a very general term that implies **the declaration of class B being visible when you declare class A.**

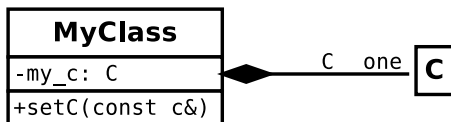
Maybe just because a method of A takes an object of type B as parameter:

```
class A
{
    ....
    // I need at least a forward declaration of B
    void fun(const &B b);
    ....
}
```

# Composition

A basic but probably the most important collaboration is that of **composition**, obtained by simply having an object of another class as member. Often the composed object is kept **private**, and is used by methods of the class.

In composition, **the lifespan of the composed object coincides with that of the composing object**. In other words, the composer "owns" its components (sometimes referred as "resource").



Here, **MyClass** is composed with one **C**.

# Composition with polymorphic objects

What happens if I want to compose my class with a polymorphic object? The cleaner solution is to use a unique pointer. After all, it implements the "unique ownership" we need! For instance

```
class MyClass{
public:
    // A constructor that moves the polygon
    MyClass(std::unique_ptr<AbstractPolygon> p):my_p{std::move(p)}{}
    // example of a setter
    void setPoly(std::unique_ptr<AbstractPolygon> p){my_p.reset(p);}
    ...
private:
    std::unique_ptr<AbstractPolygon> my_p;
    ..
}
```

std::move() is used to move the pointer, the method reset() replaces the pointer, deleting the possibly stored polygon. That's look fine. **But...Houston, we have a problem!**

# The class is not copy-constructible/assignable

MyClass is move-constructible and move-assignable (details in the lecture on move semantic). But **is not copy-constructible nor copy-assignable**, since unique pointers cannot be copied.

```
MyClass c{std::make_unique<Square>()};  
//so far fine, c is composed with a Square  
MyClass b=c; // ERROR!
```

The copy construction of b fails!. Syntax error!

We should strive to have copy-constructible and copy-assignable classes as far as possible. How can we do it? We need a **deep copy**: I do not want to copy the pointer, but the object pointed to!. But I do not know which concrete object I have.



## Towards the solution

I can think to create my own copy constructor and copy-assignment operators that perform a **deep copy**. In the situation of the previous slide, it has to copy a Square object.

**But in the class I do not have a Square object! I have a (unique) pointer to AbstractPolygon!** I might use **dynamic\_cast** to identify the correct sibling, but it is inefficient and not scalable at all, since I need to test for all members of the Polygon family!. Which may grow if I decide, for instance, to add Dodecahedron to it!

The solution is more subtle, and we have to introduce **clonable classes** and the so called virtual construction technique.

## A closer look at the problem

To make the problem clearer, if I dereference a pointer to a base class I **obtain an object of the base class**. So even if `my_p` actually points to a `Square` object: `*my_p` provides an `AbstractPolygon` object, not a `Square`.

Moreover, in this case, I will also get a compiler error since a pointer to an abstract class cannot be dereferenced: I cannot create objects of an abstract class.

But this is not relevant here, the problem of how to copy the stored `Square` object is present even if `AbstractPolygon` were a concrete class.

A clean solution is in the next slide.

## Clonable classes (virtual constructor)

It would be nice if a polymorphic object were able to **clone** itself. This can be obtained by the so called **Prototype** design pattern. Let's apply it to our Polygons:

```
class AbstractPolygon{ // base class  
    ...  
    protected: // but public is also ok  
    virtual std::unique_ptr<AbstractPolygon> clone() const=0;  
};
```

```
class Square: public AbstractPolygon{  
    ...  
    protected: // but public is also ok  
    std::unique_ptr<AbstractPolygon> clone() const override  
        {return std::make_unique<Square>(*this);}  
};
```

std::make\_unique<Square>(\***this**) initializes a new Square with a copy of the current object (\***this**) (which is a Square). The resource is handled by a unique pointer to Square, which is then converted to a unique pointer to AbstractPolygon (it is possible since Square derives from AbstractPolygon), and returned.

## a use of clonable classes

Now I can do

```
class MyClass
{
    public:
        // copy constructor
        MyClass(MyClass const & c):my_p{c.my_p->clone()}{};
        // assignement
        MyClass & operator = (MyClass const & c)
        {
            if (&c != this) my_p=c.my_p->clone();
            return *this;
        }
        // but I have to bring back move operators!
        MyClass(MyClass&&)=default;
        MyClass & operator = (MyClass&&)=default;
};
```

Et voilà: the class is copy constructible/assignable and copy operations make a deep copy. Houston, problem solved!

## Can you do better?

**Yes, you can.** You can create a new type of unique pointer, which, provided that the pointed object is clonable, has a copy constructor and an assignment operator that do a deep copy. So you do not need anymore to define your own copy constructor and the assignment operator for MyClass. All copy/move synthetic operators are fine!

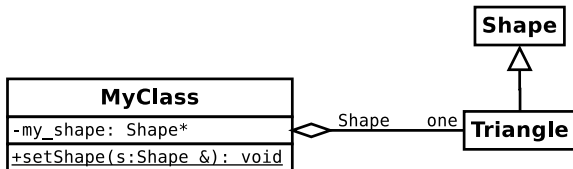
This is a better design since it obeys the single responsibility principle: it is the responsibility of the pointer owning the Square to copy it properly!

See the implementation of the PointerWrapper class in [Utilities/CloningUtilities.hpp](#), where you find also a home-made type trait and related concept that verifies if a class has the method clone... And you have in the same directory a test program.

# Aggregation

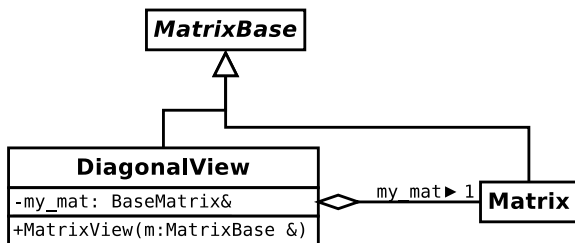
A second important type of collaboration is **aggregation**. The class stores a (classic, non-owning) pointer or a reference to a (possibly) polymorphic object **external to the class**. In the case of a reference, the latter must be initialized in the constructor and obviously you cannot reassign it.

The lifespan of the referenced object is independent from that of the containing one, and the former should outlive the latter. If the aggregated object is used to define part of the implementation of the class it 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 members of the aggregating object using a different (usually more specialized) interface, for a particular use. For instance, accessing a general matrix as a diagonal matrix. It is a particular instance of the [decorator](#) design pattern.



## Implementation details

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

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

This is a working and simple view. More sophisticated and general views require some care, we will see it in the lecture about templates.



# Friendship

A class can be declared **friend** of another class. This way the friend class may **access its private member**.

```
class A{  
    ...  
    friend class B;  
private:  
    double x;  
};  
class B{  
    void f (A & a)  
    { a.x=10.; // I can do it!  
    }  
};
```

**Friendship is a strong relationship:** a friend class may be considered as part of the implementation. Use it only when necessary.

# Friend functions

Another type of friendship is made with functions. It gives those functions the possibility to access the private member of the friend class. Sometimes it is used to speed-up access or to do special operations that require access to the private member, a common example is the streaming operator.

```
class A{  
...  
friend std::ostream & operator<<(std::ostream & out, A const &);  
...  
};  
// now operator << can access private members of A.
```

Use only if necessary. And remember: "friends of my parents are not my friends" applies also to classes, friendship is not inherited.

## Some advice

The general design of a code follows normally a top-down approach. Starting from the final objective, you identify the tasks to reach that objective.

But programming then follows a bottom-up approach. Each basic task of your code, or set of closely related tasks, should be enucleated in a class, **that you test separately.**

Only after having verified the single components, you compose them in the class (or set of classes) that implements your final objective.

Whenever possible, try to make components that can be reused and avoid code duplication.

# Pointer or reference?

Aggregation may be obtained either with pointer or with reference.  
Which one to use?

- ▶ Use reference when the aggregated object does not change. Typically, this is the case in a View;
- ▶ Use pointers if the aggregated object may be changed run time;
- ▶ If you use a reference the aggregated object must be passed by the constructor (unless it is a global variable, but this case is very rare). So your class cannot be default-constructable;
- ▶ If you use a pointer, **always initialize the pointer to nullptr** and create a method to test whether it has been assigned to an object. (initialising pointers to nullptr is always a good thing).

# Setters and getters

Methods that give access to a data member are called getters and setters and often you find them in this form

```
class A{  
  public:  
    auto get_x() const {return x};  
    void set_x(const std::vector<int> & x_){x=x_;}  
  private:  
    std::vector<int> x;  
};
```

But, really. this looks more Java than C++. Let's make some considerations.

# Setters and getters

Setters/getters are necessary if

- ▶ You plan in the future to change the internal representation of the data member, maintaining however the public interface.
- ▶ Setters and/or getters perform also other operations than just "setting" and "getting" the data member.

If not, and your setter and getter just provide full access to your data member (as in the previous example), isn't it much simpler to leave the data public?

```
class A{  
public:  
    std::vector<int> x;  
};
```

There is nothing wrong in having data public, if that is what you need. Keep instead private/protected data members that are just used by the class implementation or that you may plan to change.

# Getter and Setters

Moreover in C++ there is a more C++-style way to implement setters and getters (and the setter is also more elegant)

```
class A{  
public:  
auto x()const {return x_};  
auto & x(){return x_;}  
private:  
std::vector<int> x_  
};  
...  
auto x = a.x(); //getter  
a.x()=std::vector<int>{3,4,5}; //setter
```

If you just want a getter, implement only the const version.  
I find this more C++ style, even if I sometimes use the other version only because provided automatically by Eclipse.

# Traits

Traits are concepts mostly used in the context of generic programming. However, they may be useful also in general, so we give a preview here (more details in another lecture).

**Traits** are class (normally implemented as **struct**, but sometimes just namespaces) that provides the main **types** used in a code.

Let's make an example: I want to build a class that represents the concept of a function  $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}$ , for instance to represent a cost function in a minimization algorithm.

I need to decide how to represent the argument and the return value, as well as the function itself. It is useful to encapsulate this information in a trait. Safer and more flexible.



# An example of trait

```
struct NonLinSysTraits{  
  using Scalar=double; // the basic type for a real number  
  using ReturnType=Scalar; // the type for the image of the function  
  using ArgumentType=std::vector<Scalar>; // the type for the argument  
  // The type used to store the function  
  using Function=std::function<ReturnType (Argumenttype const &)>;  
};
```

In my code I will extract the type alias from the trait and use them consistently

```
class NonLynSys{  
  using ReturnType=NonLinSysTraits::ReturnType;  
  ... //etc.
```

or simply by using inheritance

```
class NonLynSys: public NonLinSysTraits  
{  
  // I have all types inherited!  
}
```

# The advantages of using the trait technique

- ▶ I have all main types in the code defined in a single place with significant names. In a complex code with several components, it is now easier to be consistent and avoid silly mistakes.
- ▶ A user finds the definitions of the main types in a single place: better readability.
- ▶ If I decide to change something, for instance using a vector of the Eigen library instead of `std::vector` as `ArgumentType` I just have to change one line. Other modifications in the code may be required, but at least for what concerns the type it is all set.

Traits are a nice technique, particularly when dealing with complex code. In the context of generic programming they become even more useful, as we will see.

## An example of use of [this]

With **this** we get the **this** pointer to the calling object!

```
class Foo{
public:
    void compute() const;
private:
    double x_=1.0;
    vector<double> v_};
... // definition
void foo::compute(){
    auto prod=[this](double a){x_*=a;};
    std::for_each(v_.begin(), v_.end(), prod);
...
Foo myFoo;
...
auto res=myFoo.compute();
```

Here compute() uses the lambda prod that **changes** the member x\_.  
To be more explicit you can write **this**—>x\_\*=a;.

## An example of use of `[*this]`

With `*this` we get a **copy** of the calling object! Here generic capture is handy:

```
class Foo{
public:
    void compute() const;
private:
    double x_=1.0;
    vector<double> v_};
... // definition
void foo::compute(){
    auto prod=[foo=*this](double a){return foo.x*a};
    std::for_each(v_.begin(), v_.end(), prod);}
...
Foo myFoo;
...
auto res=myFoo.compute();
```

Now, in the execution of the last statement, `foo` inside the lambda is a **copy** of `myFoo`.