15-6-2017

Class

- **Operators**
  - As a rule of thumb is better to minimize the number of operators declared in a class type limiting them to the ones that operates on the representation, i.e. data members:

```
class simple {
  public:
        simple(int a=0, int b=0): a_(a), b_(b) {};
        ~simple() { std::cout << " Ciao ciao " << std::endl; }
        int Get_a(){ return a_;}
        int Get_b(){ return b_;}
        void operator+(simple not_a) {
              a_ =a_ +not_a.Get_a();
              b_ =b_ +not_a.Get_b();
        }
  private:
        int a_,b_;
};
```

Will work?

Class

- Operators:

```
class simple {
  public:
        simple(int a=0, int b=0): a_(a), b_(b) {};
        ~simple() { std::cout << " Ciao ciao " <<
  std::endl; }
        int Get_a(){ return _a;}
        int Get_b(){ return _b;}
        void operator+(simple not_a) {
              a_ =a_ +not_a.Get_a();
              b_ =b_ +not_a.Get_b();
        }
  private:
        int a_,b_;
};
```

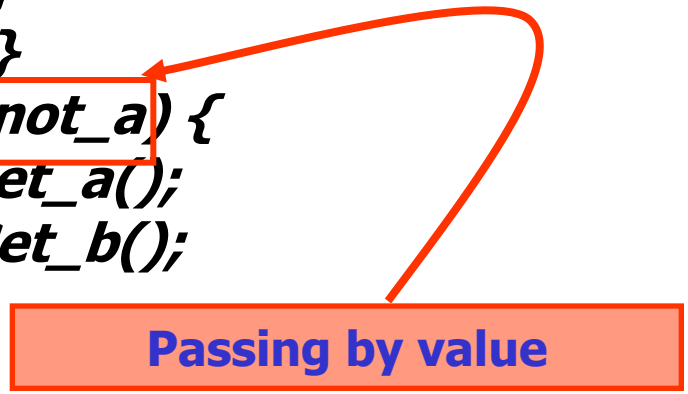
Returning a void !!

- Can I write: *simple a,b; simple c=a+b;* ?

Class

- Operators:

```
class simple {
  public:
        simple(int a=0, int b=0): a_(a), b_(b) {};
        ~simple() { std::cout << " Ciao ciao " <<
  std::endl; }
        int Get_a(){ return _a;}
        int Get_b(){ return _b;}
        void operator+(simple not_a) {
              a_ =a_ +not_a.Get_a();
              b_ =b_ +not_a.Get_b();
        }
  private:
        int a_,b_;
};
```



- Do I really need a copy ?

Class

- Operators:

```
class simple {
  public:
        simple(int a=0, int b=0): a_(a), b_(b) {};
        ~simple() { std::cout << " Ciao ciao " <<
  std::endl; }
        int Get_a(){ return _a;}
        int Get_b(){ return _b;}
        void operator+(simple not_a) {
              a_=a_+not_a.Get_a();
              b_=b_+not_a.Get_b();
        }
  private:
        int a_,b_;
};
```

Modifies data members!!!

- It is really a *+* operator? Does *+* modify the object is applied on?

Class

- Operators:
  - A *+* operator, must return a temporary object and not modify the one is applied on.

    c=a+b; // neither a or b are modified
  - A *+=* operator, does modify the object is applied on. It must return a reference to the object is applied on.

    ++(a+=b); // (a=(a+b),a=a+1)
  - A canonical form for both operators:

```
T& T::operator+=(const T& a) {
// …
   return *this;
}
Const T operator+(const T&a, const T&b) {
   T temp(a);
   temp+=b;
   return temp;
}
```

Helper function!
Not declared in the class definition

Class

- **Operators**
  - As a rule of thumb is better to minimize the number of operators declared in a class type limiting them to the ones that operates on the representation, i.e. data members:

```
class simple {
    public:
            simple(int a=0, int b=0): a_(a), b_(b) {}; // Constructor
            simple(const simple &a):a_(a.a_), b_(a.b_) ; // Copy constructor
            ~simple() { std::cout << " Ciao ciao " << std::endl; } // Destructor
            int Get_a() const { return _a;}
            int Get_b() const { return _b;}
            void Print() const {std::cout << " Simple : a " << _a
                        << ", b " << _b << std::endl; }
            simple& operator+=(const simple & not_a) {
                        std::cout << " Sono in simple += simple" << std::endl;
                        a_+=not_a.Get_a(); b_+=not_a.Get_b(); return *this; }
            simple& operator+=(int a) {
                        std::cout << " Sono in simple += int" << std::endl;
                        a_+=a; b_+=a; return *this; }
    private:
            int a_,b_;
};

const simple operator+(const simple &, const simple &);
const simple operator+(const simple &, int);
```

simple.hpp

Class

- Operators

```
const simple operator+(const simple &a, const simple &b) {
  simple temp(a);
// Same as return temp.operator+=(b);
  return temp+=b;
}
const simple operator+(const simple &a, int b) {
  simple temp(a);
  return temp+=b;
}
```

simple.cpp

  - Note that the code snippets are generic. They just rely on the fact that a copy constructor and operators *+=* are defined

Class

```
#include <iostream>
#include "simple.hpp"

simple change_it_ref( simple & a) {
  return a+=1;
}

simple change_it_cop( simple a) {
  return a+=1;
}

int main(){
  int a,b;
  simple c,d,e;

  while(std::cin >> a) {
    b=a-1;
    c.Set_a(a);
    c.Set_b(b);
    c.Print();
    d+=c;
    d.Print();
    // Arghhhhhhhhh memory leak !!!
    new simple(d);
    e=c+d;
    e.Print();
    e=change_it_ref(d);
    e=change_it_cop(d);
  }

  simple f=d;
  f=d;
  simple g(d);
  return 0;
}
```

1. `g++ -c simple.cpp`
2. `g++ -c test_simple.cpp`
3. `g++ -o test_simple simple.o test_simple.o`
4. `./test_simple`

Compilation

Link

Execute

test_simple.cpp

- Test_simple.tgz (https://owncloud.ba.infn.it/public.php?service=files&t=a135b73861c6bb008e7dc795f8f06e6f)

Exercise

- Build the post and pre increment operator (++), for the class: simple

Exercise

- Let's build the preincrement:

```
simple operator++(){
    cout << "Sono in simple ++ prefisso " << endl;
    ++a_; ++b_;
    return *this;
  }
```

Must return a reference to permit: *(++a)--;*

- Both operators share the same symbol: *++*, how can we distinguish them?
  - The postincrement operators is declared with a dummy *int* argument:

return_type operator++(int){…};

```
simple operator++(int){
    cout << "Sono in simple ++ postfisso " << endl;
    simple temp=*this;
    ++(*this);
    return temp;
}
```

Will work?

Must return a const value to avoid: *a++++;*

Exercise

- Now it works:

```
simple & operator++(){
   std::cout << "I am in the simple preincrement operator ++ " << std::endl;
   ++a_; ++b_;
   return *this;
 }

const simple operator++(int){
   std::cout << "I am in the simple postincrement operator ++ " << std::endl;
   simple temp=*this;
   // Always implement postincrement in term of preincrement
   ++(*this);
   return temp;
}
```

Class

- Conversion operators, i.e. implicit conversions
  - It is possible to define operators that are able to convert (cast) our user defined type to another type.
  - Instead of the operation symbol, in the operator name the name of the type you want to convert to must be used:

```
Class X{
        private:
         …
        public:
        …
        operator T() const { // do some conversion from X to T }
        …
};
```

Class: explicit

- *explicit*
  - A constructor having a single formal parameter could be interpreted as a casting operator:

```
class complex{
public:
        complex(double); // it can be used to create a complex using a double
        complex(double, double);
        …
};
complex c=3.4; // right define the complex object (3.4,0)
```

  - To disable this behavior and force the constructor to be interpreted just as a constructor, not as an implicit conversion operator, the *explicit* prefix must be used.

```
        explicit complex(double); // No implicit conversions
```

Exercise on classes

1. Write a class describing a Date, *i.e.* day month year, and corresponding operators.
2. Write a class describing a specialized pair: angle and its cosinus. Assigning angles must automatically corresponds a cosinus assignment and viceversa. The class must be written:
   - Without using an *if statement;*
   - Using only STL *iostream* (in case is needed);
   - Trigonometric functions are declared in the header: *cmath*
     - *double cos(double x) ; // return the value of a cosinus calculated for x radiants*
     - *double acos(double x); // returns the principal value of the arccosines of x, in radiant. x must be inncluded in the range [-1,1]*

Exercise

```
.
.
.
// Create a specialized pair (angle+cos)
Pair_of_angle_and_cos  a;
// Create and angle
Angle b=45;
// assign the angle to the pair
a=b;
// or
a.SetAngle(b);
// get the cosinus of the angle
double cos45=a.GetCos();
// Create a cosinus
a.SetCos(1.);
// Get the corresponding angle
double d=a.GetAngle();
Angle ad=a;
// Set a new cosinus
Cosinus cd=a.GetCos();
.
.
.
```

Exercise

- We need two types: an angle, a cosine.
- It should be possible to assign a double to both types.
- It should be possible to treat them as doubles, i.e. to implicitly cast them to double.
- It should be possible to assign an angle to a cosine and vice versa.

Exercise

```
class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t){ang=acos(t); return *this; }
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};
```

Ang t(90.);

Ang t(90.), c(t);

Ang t(90.); double a=t;

Ang t; double a=90.; t=a;

Ang t; CAng a=0.5; t=a;

CAng t(.5);

CAng t(.5), c(t);

CAng t(.5); double a=t;

CAng t; double a=.5; t=a;

CAng t; Ang a=90.; t=a;

Exercise

```
class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t){ang=acos(t); return *this; }
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};
```

Will work ?

Exercise

```
class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t) {ang=acos(t); return *this; }
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};
```

I need to declare the name *CAng* before using it!

Exercise

```
class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t){ang=acos(t); return *this; }
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};
```

Remember. We can declare! So we can introduce just a name, also for a class. This is called a forward declaration.

Exercise

```
#include <cmath>

class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_=t; return *this; }
  Ang & operator=(const CAng &t){ang_=acos(t); return *this; }
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_=t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_=t; return *this; }
  CAng & operator=(const Ang &t) { cang_=cos(t); return *this; }
};
```

Will work ?

I need to include the declarations of acos & cos before using them!

Exercise

```
#include <cmath>

class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_=t; return *this; }
  Ang & operator=(const CAng &t){ang=acos(t); return *this; }
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_=t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_=t; return *this; }
  CAng & operator=(const Ang &t) { cang_=cos(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=acos(t); return *this; }
```

Will work ?

… were I can write this operator definition?

The implicit cast must be definited!
ang=acos(static_cast<double>(t));

Exercise

```
#include <cmath>

class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t);
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=acos(t); return *this; }
```

Will work ?

It is tollerated because we are in the same class definition.

Accessing a private data member!!!!

Exercise

```
#include <cmath>

class CAng;

class Ang {
private:
  double ang_;
public:
  Ang( const double &t=0.) : ang_(t) {};
  Ang( const Ang &t) { ang=t.ang_; }
  operator double() const { return ang_; }
  Ang & operator=(const double t) { ang_ =t; return *this; }
  Ang & operator=(const CAng &t);
};

class CAng {
private:
  double cang_;
public:
  CAng( const double &t=0.) : cang_(t) {};
  CAng( const CAng & t) { cang_ =t.cang_; }
  operator double() const { return cang_; }
  CAng & operator=(const double t) { cang_ =t; return *this; }
  CAng & operator=(const Ang &t) { cang_ =cos(t); return *this; }
};


Ang & Ang::operator=( const CAng &t) { ang=acos(t); return *this; }
```

Will work ?

YES!!!

Exercise

- Having these two classes is now possible to write:

```
Angle angl;
CAngle cosin;
double val=45.;
// angl.operator=(static_cast<double>(cosin.operator=(val)))
angl=cosin=val;
```

Exercise

```
class Angles {
public:
  typedef Ang Angle;
  typedef CAng CAngle;

  Angles() { SetAngle(0);  }
  Angles(const CAngle & ct) { SetCAngle(ct); }
  Angles(const Angle & t) { SetAngle(t); }
  Angles(const Angles & a) { SetAngle(a.GetAngle()); }

  Angle GetAngle() const { return angle_; }
  CAngle GetCAngle() const { return cangle_; }
  void SetAngle( Angle t ) { cangle_=angle_=t; }
  void SetCAngle( CAngle ct ) { angle_=cangle_=ct; }

  Angles & operator= ( const Angles & a) {
    SetAngle(a.GetAngle());
    return *this;
  }

  Angles & operator= ( const Angle &a) {
    SetAngle(a);
    return *this;
  }

  Angles & operator= ( const CAngle &a) {
    SetCAngle(a);
    return *this;
  }

  void Print( std::ostream & o=std::cout) const {
    o << " Angle: " << angle_ <<", Cosine: " << cangle_ << std::endl;
  }

private:
  Angle angle_;
  CAngle cangle_;

};
```

Exercise

```
#include <iostream>
#include "angles.h"

const double GRAD2RAD=acos(0.)/90.;

int main() {
  double t;
  Angles::Angle a;
  Angles b;
  std::cout << " Enter an angle (Degree):" ;
  while(std::cin >> t) {
    a=GRAD2RAD*t;
    b=a;
    b.Print();
    std::cout << " Enter an angle (Degree):" ;
  }
  return 0;
}
```

- test_angle.tgz, https://owncloud.ba.infn.it/public.php?service=files&t=dd78377687db39ec48e7006924c3789b

Class

- Derived classes
  - It is possible to express common properties between classes?
  - Classes should express concepts. Are these concepts underlying to different class definitions, related or common?

```
struct studente {
    string name;
    int anno_iscrizione;
    int numero_matricola;
    vector<string> esami_sostenuti;
};
struct studente_dottorato {
    studente sd;
    int ciclo_dottorato;
    vector <string> classi_dottorato;
};
```

Class

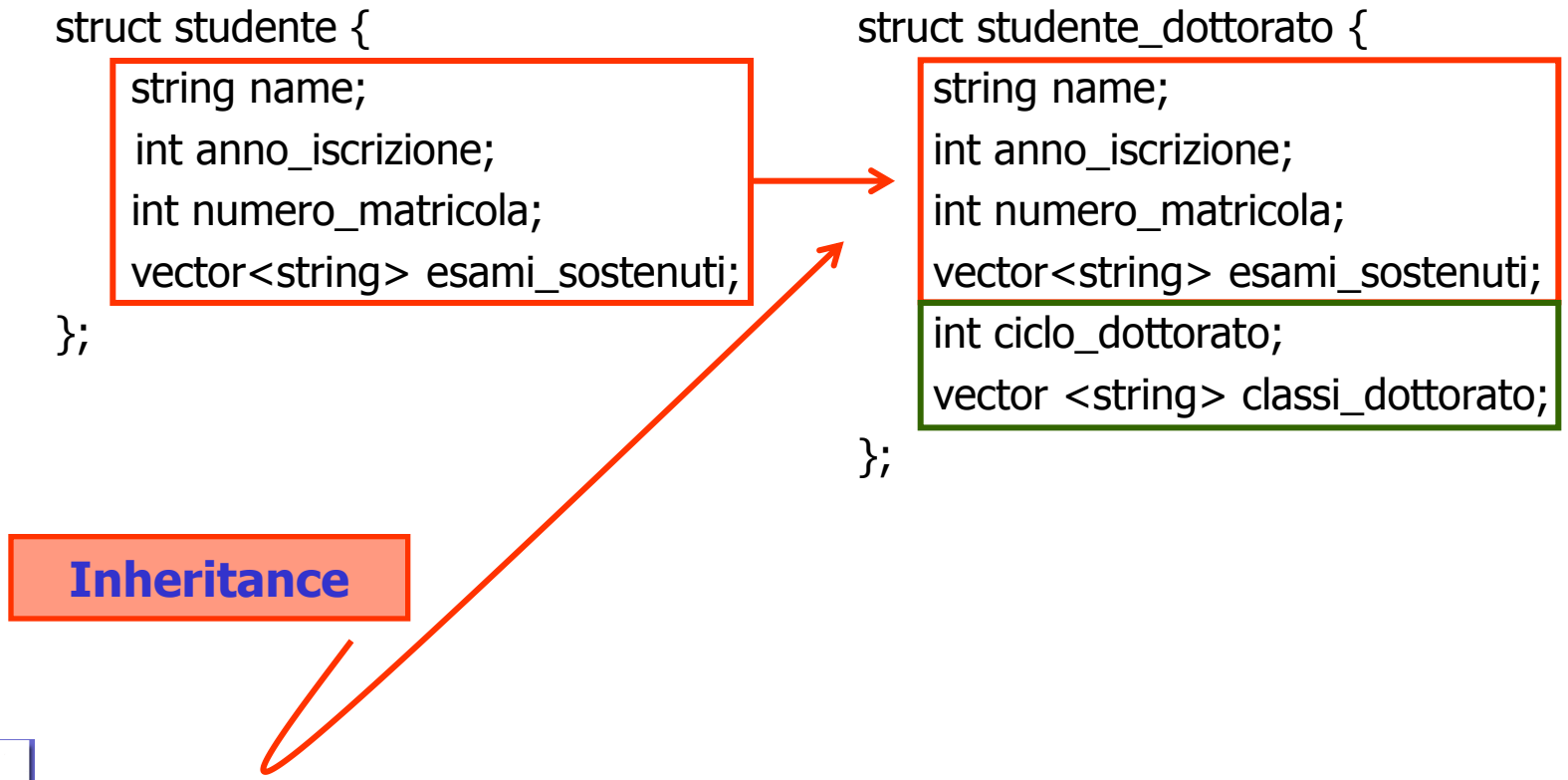
- Actually is it a *studente di dottorato* a *studente* with a bit more attributes?
- Can we state that a *studente di dottorato* is a s*tudente* storing a little bit of information more?

```
struct studente_dottorato: public studente {
    int ciclo_dottorato;
    vector <string> classi_dottorato;
};
```

- This new statement instruct compiler about the fact that *studente di dottorato* is a derived class from *studente*.
- *studente* became a base class for *studente di dottorato*.
- A derived class inherits the base class proprierties. This relationship is called: inheritance

Class

- Imagine a derived class as a base class object to which more information are added at the end:



Class

- Using a class as a base one it is so equivalent to a declaration of an object wihout the given name:

```
class studente;                // può essere incluso in un header file
struct studente_dottorato: public studente {
   ...
};
```

It is equivalent to the creation of an unamend *studente* object

- But why I should use such a messy technique if I can just include a class object as a class member? E.g. :

```
struct studente_dottorato {
   studente sd;
   int ciclo_dottorato;
   vector <string> classi_dottorato;
};
```

Class

- Using inheritance the derived class is considered a subtype of the base class.
- It can be used anytime a base class is used:

```
vector <studente *> altogether;
void add_student( studente * s) {
   altogether.push_back(s);
}
…
int main(){
   studente laurea;
   studente_dottorato dott;
   …
   add_student(&laurea);
   add_student(&dott);
   return 0;
}
```

Using *studente_dottorato* as it is a generic *studente*

Class

- Contrary is not true. I have to use an explicit cast.

```
...
int main(){
   studente laurea;
   studente_dottorato dott;
   studente *ps=&dott;
   ...
   add_student(&laurea);
   add_student(&dott);
   std::cout << static_cast<studente_dottorato *>(ps)->ciclo_dottorato
   << endl;
   return 0;
}
```

Using *studente_dottorato* as it is a generic *studente*

Using generic *studente* as it is a *studente_dottorato*. IS NOT LEGAL!!!

Explicit cast is needed

- An object of the derived class type, can be treated as one of the base class, using pointers and references.

Class

- Member functions
  - What happens to the member functions of the two class type?
  - On which members they could act, i.e. which data can they modify?
  - A derived class function member can use all members declared as: public or protected, in its base class definition. Exactly as they would be declared in its *class definition*.
  - Even a member of a derived class cannot use any member declared as private in the class base definition.

Class

- *protected*
  - Any protected member can be used by any of the derived classes members.
  - For all other entities (functions, *helper functions,* different type objects, class using them as memberts etc. etc. etc.) they will result private, not modifiable by them.

Class

It is a private member of the class studente

Where is the error?

```
class studente {
private:
    string   name_;
    int   anno_iscrizione_;
    int   numero_matricola_;
    vector<string> esami_sostenuti_;
public:
    void SetName(string &s){ name_=s;}
    string GetName() {return name_;}
    …
};
class studente_dottorato :public studente {
private:
    int   ciclo_dottorato_;
    vector <string>   classi_dottorato_;
public:
    void SetName(string &s){ name_=s;}
    …
};
```

Class

```
class studente {
private:      protected:
      string   name_;
      int   anno_iscrizione_;
      int   numero_matricola_;
      vector<string> esami_sostenuti_;
public:
      void SetName(string &s){ name_=s;}
      string GetName() {return name_;}
      …
};
class studente_dottorato :public studente {
private:
      int   ciclo_dottorato_;
      vector <string>   classi_dottorato_;
public:
      void SetName(string &s){ name_=s;}
      …
};
```

It must be declared as protected

Class

```
class studente {
private:
      string   name_;
      int   anno_iscrizione_;
      int   numero_matricola_;
      vector<string> esami_sostenuti_;
public:
      void SetName(string &s){ name_=s;}
      string GetName() {return name_;}
      …
};
class studente_dottorato :public studente {
private:
      int   ciclo_dottorato_;
      vector <string>   classi_dottorato_;
public:
      void SetName(string &s){ name_=s;}   { SetName(s);}
      …
};
```

It is better to use the base class *member functions*

… something else?

Class

```
class studente {
private:
      string   name_;
      int   anno_iscrizione_;
      int   numero_matricola_;
      vector<string> esami_sostenuti_;
public:
      void SetName(string &s){ name_=s;}
      string GetName() {return name_;}
      …
};
class studente_dottorato :public studente {
private:
      int   ciclo_dottorato_;
      vector <string>   classi_dottorato_;
public:
      void SetName(string &s){ studente::SetName(s); }
      …
};
```

Now works!

In case of ambiguities I have to specify the class (namespace) owing the method I like to use.

Class

- Constructors and destructors
  - What about them? How does inheritance affect them?
  - If a base class has constructors defined (so it needs initialization), these must be explicitly used with the correct parameters.

```
class studente {
public:
...
studente(string &s="none"): name_(s), … {…}
};
class studente_dottorato : public studente{
  public:
   …
  studente_dottorato(string &s="none", …): studente(s),
   anno_dottorato_(0) … { … }

};
```

Class

- Objects are constructed bottom up.
  - First the base class objects are constructed.
  - Then the class members.
  - Then the derived class.
- Objects are destroyed in the reverse order.
- Constructors are never inherited.

Class

- Copy
  - The base class know nothing about the derived class, so only the base class part will be copied in a copy of a derived class used as a base class:

```
void f(const studente_dottorato &s){
...
        studente laurea=s;
}
```

  - … same in the assignments:

```
void f(const studente_dottorato &s){
...
        studente laurea;
        laurea=s;
}
```

  - Assignments operators are not inherited as well.

Class

- Copy
  - It is a good habit to design base class having the copy constructor private.
  - Consider the case:

    *T_base *pt= new T_base(*pT_derived);*

    Here a copy constructor is used because the derived class can be used as a base class. The base class cannot know about the extra information in the derived one, so, its copy constructor, cannot treat this information at all. The derived class will be partially copied and only its "common base" part copied.
  - This can be avoided hiding the copy constructor.
  - If a copy constructor is private it can be used only by objects of the very same type.