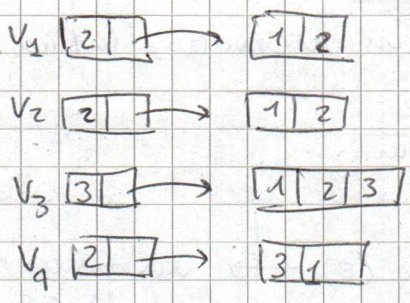
APC

Vectors :



$V_1 == V_2$

$V_1 < V_3$
$V_4 > V_1$
$V_4 > V_3$

Esercizio: Biblioteca

Book
- Code - Author - Title - Year - Pages - Available
+ Equal to (title, Author) + print()

← saranno tutte informazioni private perché i dati venerebbero cambiati dagli utenti

will print out the content

Library
- books[]
+ Add Book(b) + Rent Book(code) + Find (Author, title) + Find Available (Author, title) + Return Book (code) + print() + printoldest()

← considero dei vectors per contenere i books (sono tutti omogenei)

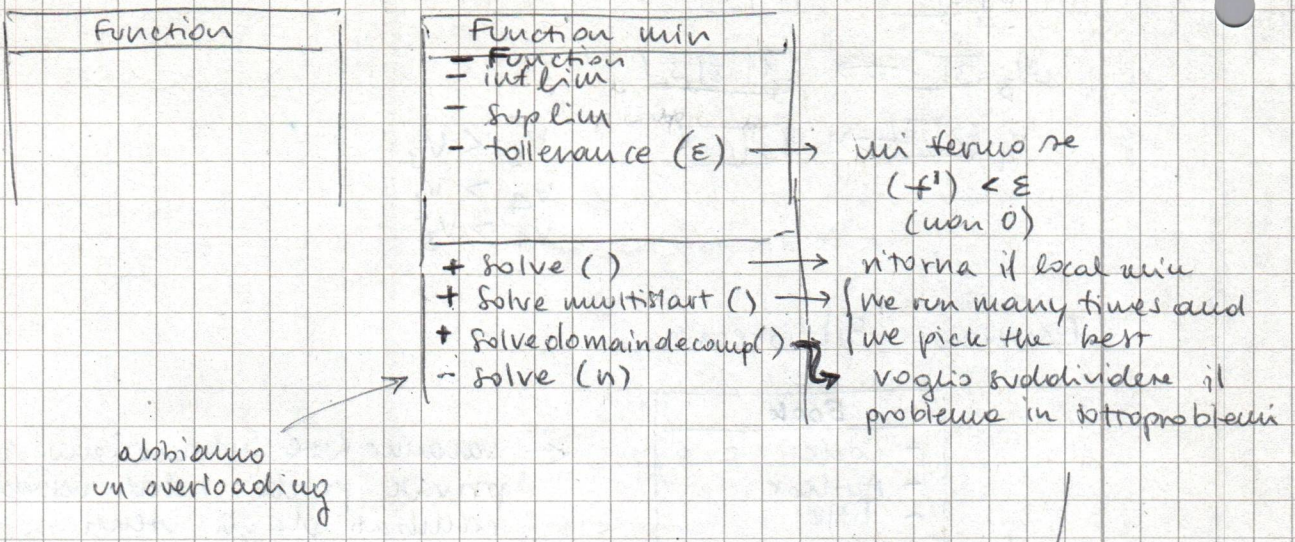
← devo prima capire se il libro c'è nella bibl.

← controllo se il libro è è disponibile

APC
8/10/2019

Gradient descent

obj: minimizzare una function su un intervallo [inflim, suplim]



solvedomaindecomp (s, k)
decompongo in s subintervals
and k trials per ciascuno

a.getX()

(*a).getX() ≡ a -> get()

(*(*a).getX()).(...) ≡ a -> getX() -> (...)

std::string

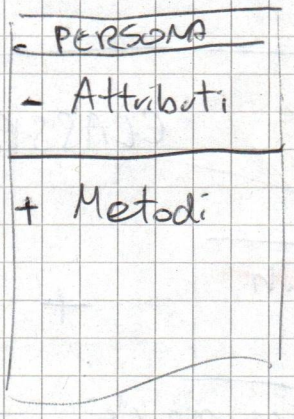
C

```
typedef struct {
    int eta;
    string nome;
    string cognome;
} persona;
```

```
persona* Paolo = malloc...
Paolo.eta = 5;
```

CPP

```
CLASSE PERSONA {
// Attributi
// Costruttore
// Metodi

private:
    int eta;

public:
    void incrementa();
    persona(int a);
    bool isMaggiorenne();
};
```




```
Persona Paolo = new Persona(5)
```

class { ... };

Altro { ... }

Ogni classe si divide in .h e .cpp

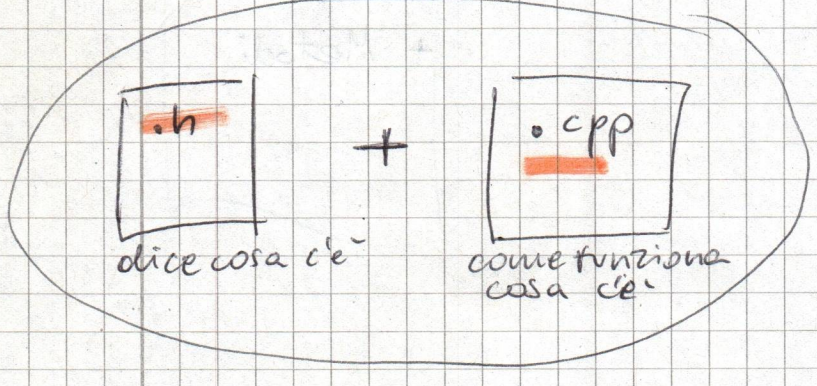
.H

```
class Persona {

private:
    int eta;

public:
    void incrementa();
    Persona(int a);
    bool isMaggiorenne();

};
```

.CPP

```
#include "persona.h"

void Persona::incrementa() {
    eta += 1;
}

Persona::Persona(int a) {
    eta = a;
}

bool Persona::isMaggiorenne() {
    if (eta >= 18) {
        return true
    }
    return false;
}
```

CLASSE: ⊛



SCOPE: le variabili esistono solo da una graffa all'altra

⊛ se voglio che isMaggiorenne non modifichi niente ⟹ ~~isMaggiorenne~~

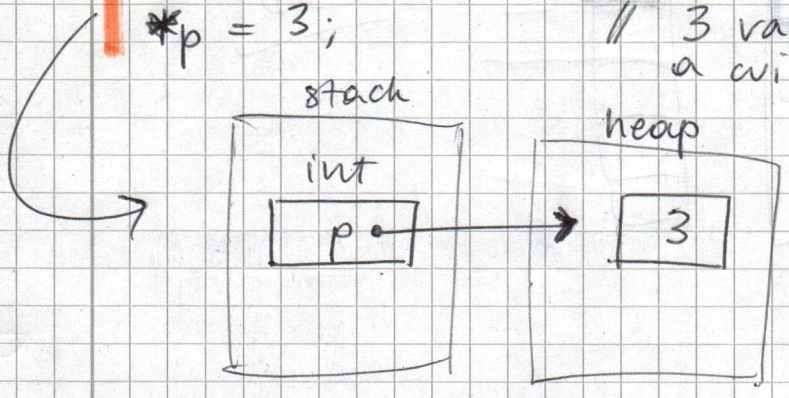
```
bool isMaggiorenne() const;
```

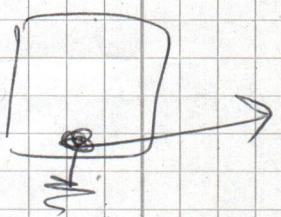
C MALLOC

```
void* malloc (int a);
```

(2 utenti): `(utente*) malloc (2 * sizeof(utente))`

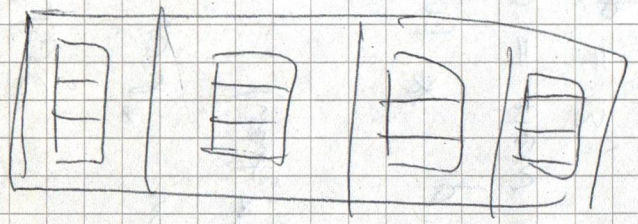
```
int * p;          // puntatore "p" a un intero
p = new int;      // crea memoria per un int nella heap
*p = 3;           // 3 va nella casella
                  // a cui punta p
```

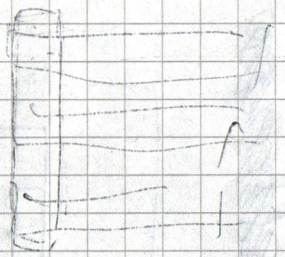




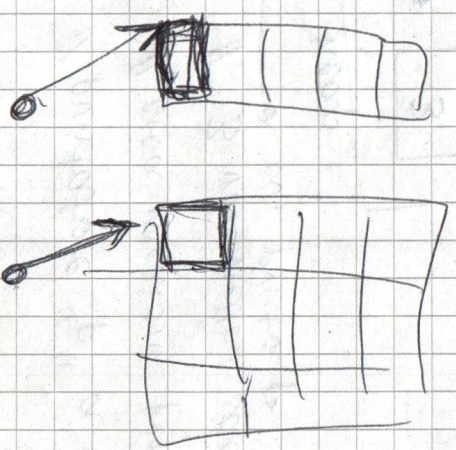
```
delete p;
delete [] p;
```

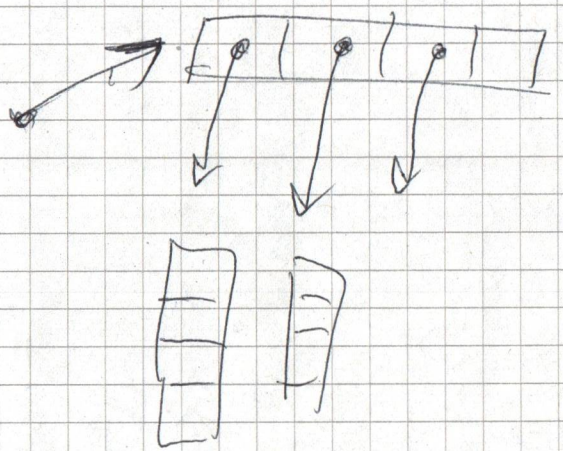
```
typedef struct {
    char nome[10];
    int eta;
} persona;

persona a;
a.nome = "Pippo";

persona* b;
b = (persona*) malloc (sizeof (persona))
(*b) -> nome
```



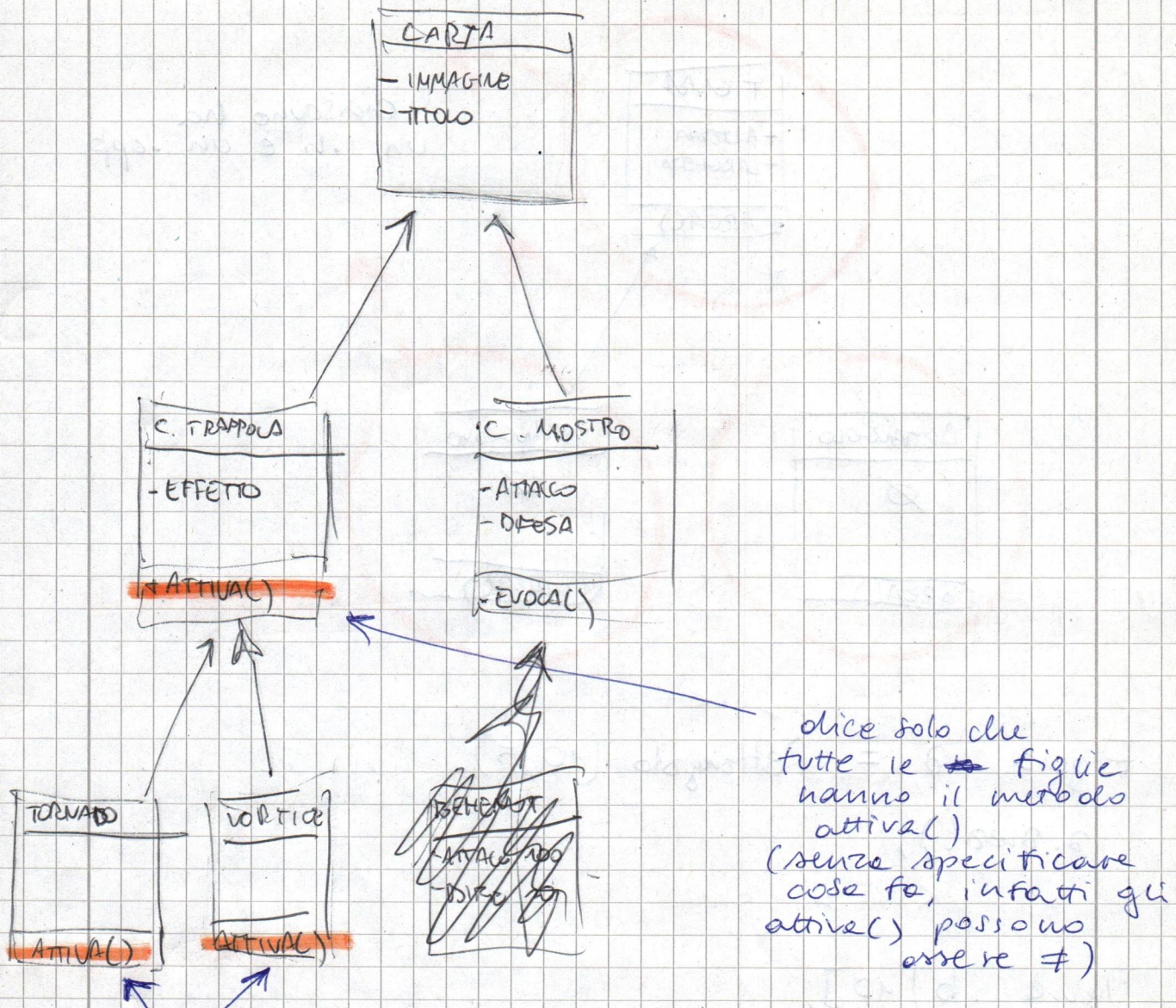


```
int a[10][10];
int** a;
```





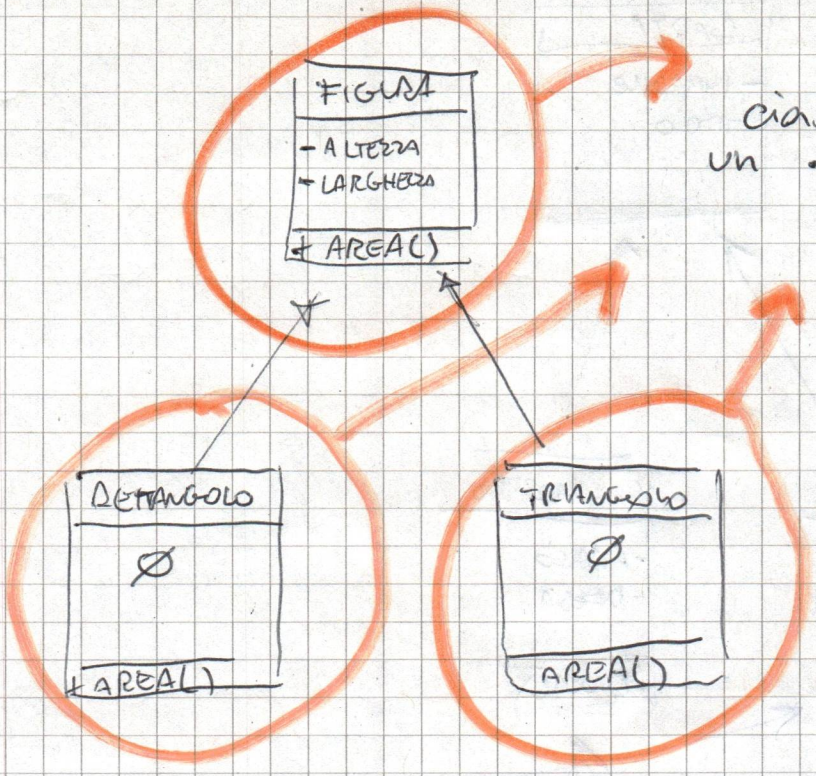
INHERITANCE



CTrappola carta1 = perdiTreppola()

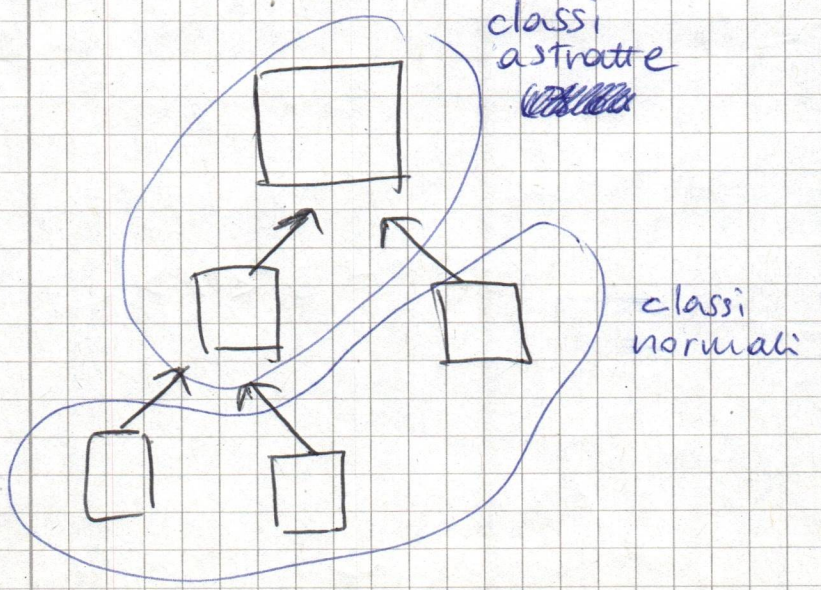
carta1.attiva();

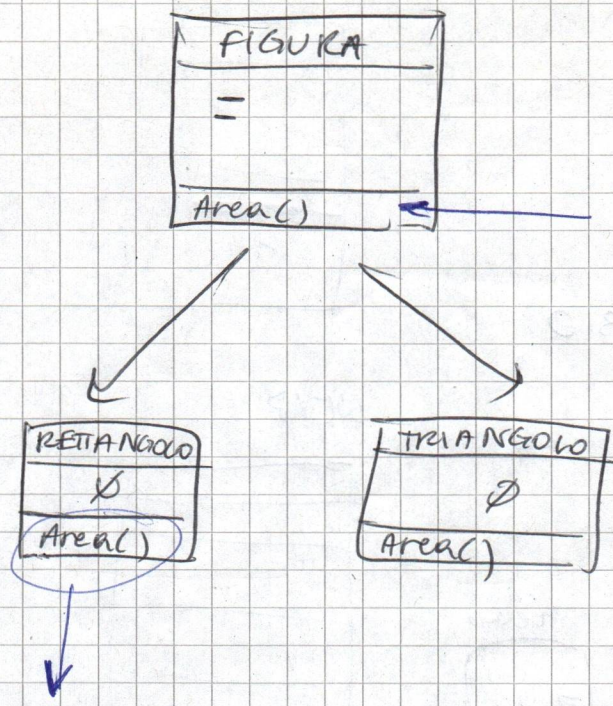
INHERITANCE PT. 2



ciascuno ha un .h e un .cpp

```
Figura a = Rettangolo (10, 5)
  a.area();

Figura* b[10];
for ... i
... + b[i].area();
```

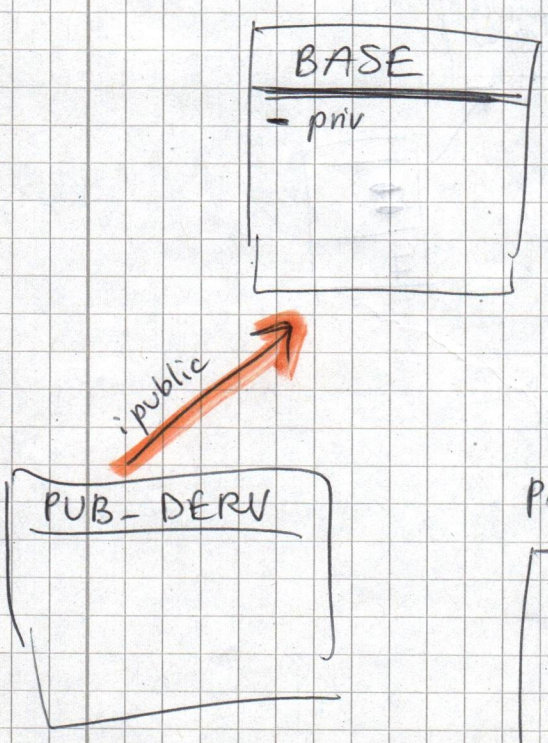




se è implementata
⇒ figura è oggettabile,
altrimenti è classe
astratta

se Figura è astratta tutto in regola,
se figura è oggettabile e voglio Area() di figura
non metto Area() in Rettangolo
se Figura è oggettabile e voglio ~~Are~~ Rettangolo.Area()
≠ Figura.Area() ⇒ devo riscriverla
(si chiama OVERRIDE)

Lez. 10 - pg. 30



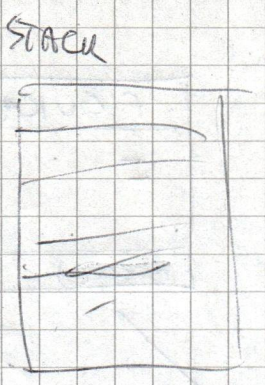
base.h

```
class Base {
  . . .
};
```

Pub_Derv.h

```
#include "base.h"
class Pub_Derv : public Base {
  . . .
};
```

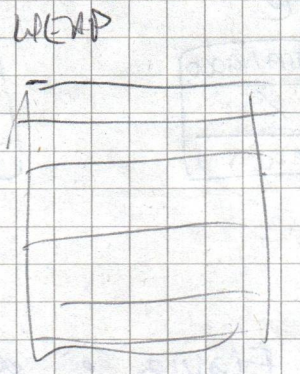
int a = 3; 3.0



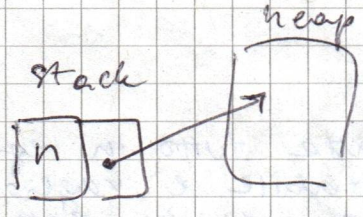
CASTING:

a → INTERO

(float) a → FLOAT 3.0



VECTORS



#include <vector> ← ce l'hanno tutti

#include "prova.h" ← lo scrivo io "prova.h"

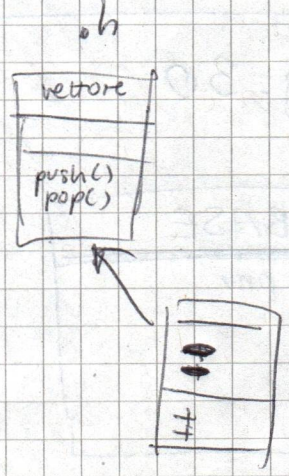
- std::vector<int> prova;

  prova.push_back(3);

  prova.size() → 1

  prova[0] → 3

  prova.empty() → 0



~~Esempio~~

int a[] = {1, 2}

std::vector<int> a = {1, 2}

```
std::string a = "ciao";
c(&(*a));

std::string* b = new std::
```

function c(std::string* ob)

n m

string

Function c (str1, str2); Temps: O(m x n)

"h" : 2 -> mm

```
class Point {
private:
    int x;
    int y;

public:
    Point(int a, int b): x(a), y(b);
};
```

main.cpp

```
Point a(3,4);
Point* b = new Point(3,4);
delete b;
```

NEW ⇒ DELETE

Function Overload

pow(4) → 16

pow(4,3) → 4^3

k opzionale

```
int pow(int n, int k){
    if(no k){
        return n*n
    }
    fo - - - -
    ris = n
```

```
int pow(int n, int k=2){
```

NON SI PUO'

```
string stampa(string a){
    return a;
}
string stampa(int a){
    return to_string(a);
```

```
int f (int a, int k);          → .h

int f (int a, int k = 3) {     → .cpp
...
}

f (3)                          → main.cpp
f (3, 4)
```

Non def. il costruttore nella classe | **Def. il costruttore**

senza costruttore

```
Screen myScreen;
```

```
class Screen {
...
public:
    Screen (int a, int b);

Screen myscreen (1, 2);
```

Costruttore std | **Costruttore def.**

```
.h
class Screen {
Private:
...
Public:
  get(...
};
```

```
.h
class Screen {
Private:
...
Public:
  Screen (int a, int b);
};
```

STACK

```
main.cpp
Screen myscreen;
myscreen.get.....
```

≠

```
main.cpp
Screen myscreen (1,2);
```

STACK

HEAP

```
main.cpp
Screen* myscreen = New Screen();
myscreen
```

≠

```
main.cpp
Screen* myscreen = New Screen (1,2);
```

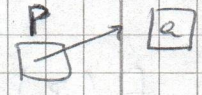
HEAP

C / ~~C++~~

```
int* a;
cosa* b;
b = (cosa*) malloc (sizeof(cosa))
```

```
typedef struct {
int el
char asd
} cosa;
```

```
int *p;
*p = e
```



REFERENCE (POINTERS ADVANCED)

```
int f(persona& a) {
    a.get(...)
```

.h

```
Class A{
  Public:
    fun();
  Private:
    Ø
};
```

main.cpp

```
A a;
a.fun();
```

STACK

```
A* a;
a -> fun();
```

HEAP

@

```
type elem1;
type* elem2;
elem2 = new type;
*elem2 = ...;
```

~~f(type &elem1);~~
~~f'(type *elem2);~~

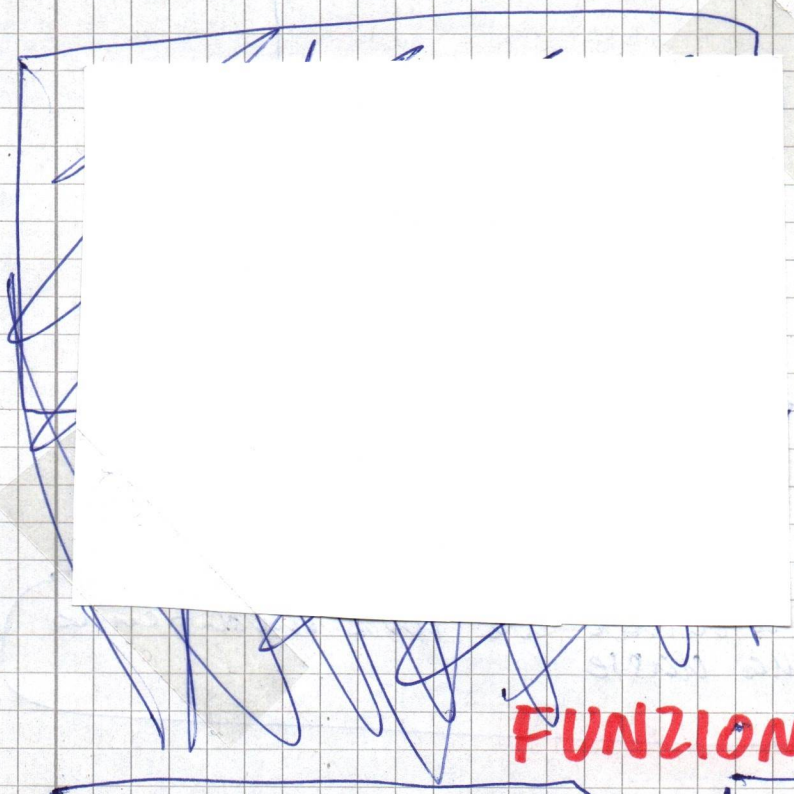
REFERENCE	POINTER
def f(type &a);	def f(type *a);
call f(elem1);	call f(elem2);
f(..){ a=...;	f(..){ *a=...;

def ~~f(type a);~~ ← è eliminato in modo che crea una copia
call ~~f(&elem1);~~

CONST :

- variabili (parametri)
- metodi (e classi) (≠ funzioni)

Se arriva un oggetto immutabile (const type var) posso usare solo metodi/funzioni immutabili



FUNZIONI

```
int print (int a);
int main(){
    const int b=3;
    print(b);
}
```

✓

```
int print (int a) const;
int main(){
    const int b=3;
    print(b);
}
```

✗

```
int print (const int a);
int main(){
    const int b=3;
    print(b);
}
```

✓

```
int print (const int a);
int main(){
    int b=3;
    print(b);
}
```

✓

METODI :/

CONST NEI METODI

METODI (! CLASSI)

~~Class Point~~ {

.h

```
Class Point {
Private:
    int x; int y;
Public:
    int getx() const {return x;}
};
```

≠ dalle functions (const ↑ prendono parametri)

con "const" sto dicendo che non modificano gli attributi della classe

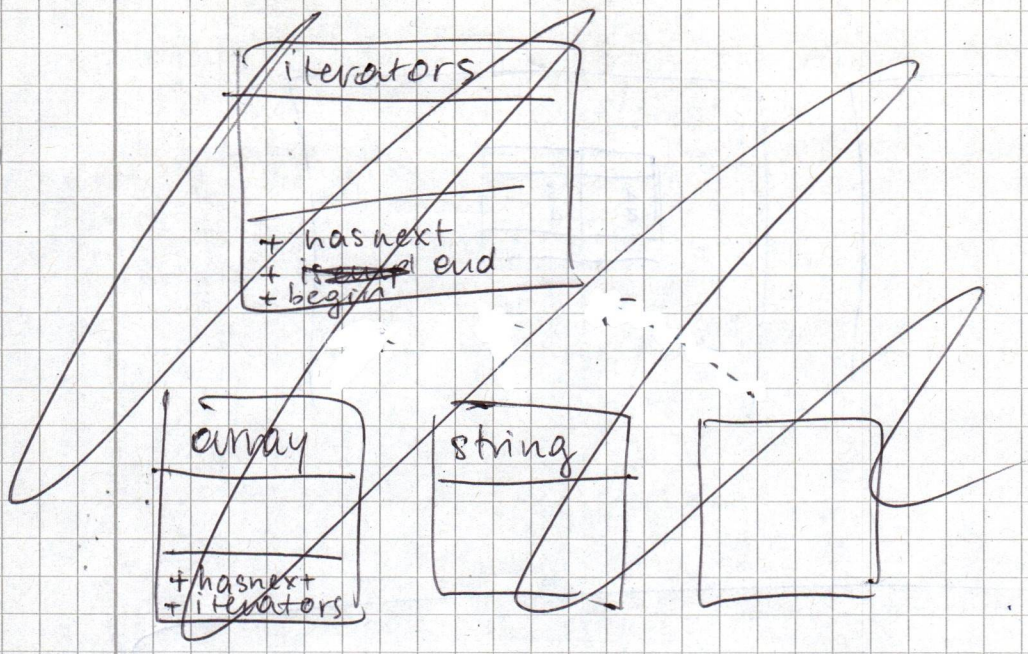
~~int main()~~ {

.h

```
Class Point {
Public:
    int getx() const { return x;}
    int gety()  { return y;}
};
```

```
int ritornacose (const Point *point) {
    return point -> getx() + point -> gety();
}
```

ERRORE
perché ho const quindi posso usare solo metodi pubblici const (getx() è const, gety() no)



ITERATORS

8'

```
int main(){
    std::vector<int> vect = {1, 2, 3, ..., 6};
    std::vector<int>::iterator it;
    it = vect.begin();
    while (it != vect.end()){
        cout << *it;
        it++;
    }
```

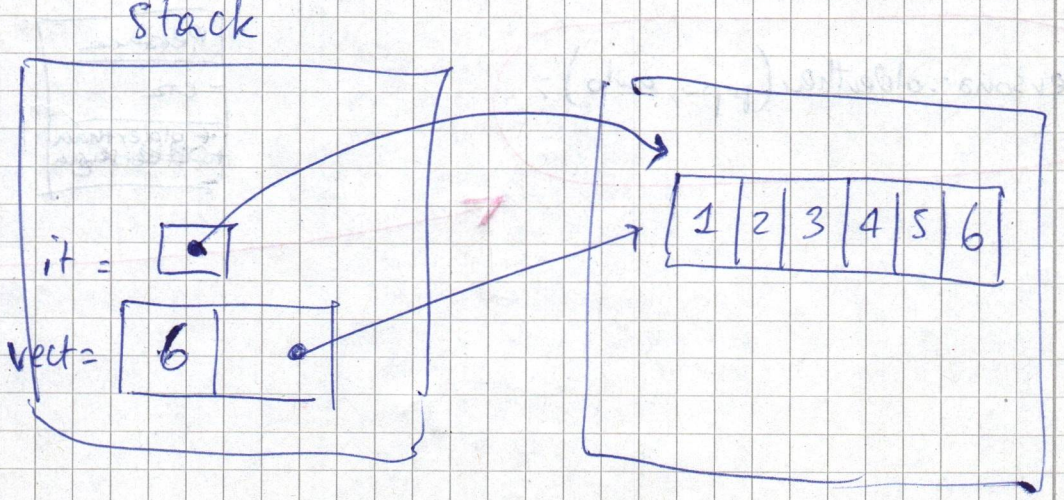
std::type::iterator it;

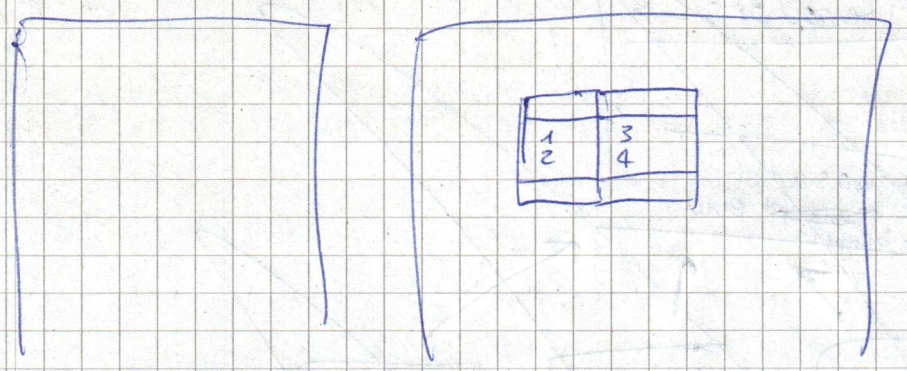
std::type::const_iterator it;

printa l'elemento a cui è arrivato it

→ legge e ~~scrive~~

→ legge (!)






```
std::vector<Point> vect;
vect.push_back(new Point(a,b));   <-- ERRORE
vect.push_back(Point(a,b));       <-- giusto

std::vector<Point*> vect;
vect.push_back(new Point(a,b));   <-- giusto
vect.push_back(Point(a,b));       <-- ERRORE
```

METODI STATICI

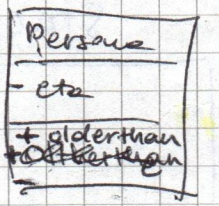
Definisco metodi in una classe
Chiamo i metodi della classe senza creare la classe

main.cpp

NON STATICA
```
pippo.olderthan(carlo)
```

```
Persona pippo(18);
Persona carlo(12);
```

STATICA
```
Persona::olderthan(pippo, carlo);
```



NON STATICA

```
class Persona {
private:
    int eta;
public:
    Persona(int a): eta(a) {};
    bool olderThan(Persona & param);
};
```

.h

```
#include "persona.h"

bool Persona::olderThan(Persona & param) {
    if (param.eta > this->eta) {
        return false;
    }
    return true;
}
```

.cpp

STATICA

```
class Persona {
private:
    int eta;
public:
    Persona(int a): eta(a) {};
    static bool olderThan(Persona & a, Persona & b);
};
```

.h

← "static" solo nel .h

```
#include "persona.h"

bool Persona::olderThan(Persona &a, Persona & b) {
    return a.eta > b.eta;
}
```

.cpp

HELPER FUNCTIONS

```
type operator<...> (..., ...){
  ⋮
}
```

← solo per classi (vector, array, struct, ...) gli int/char/... non sono classi

---

GETTER - SETTER (≡ metodi) ∈ Classi

- **getter**: legge / restituisce
- **setter**: imposta / modifica

gli attributi (-)

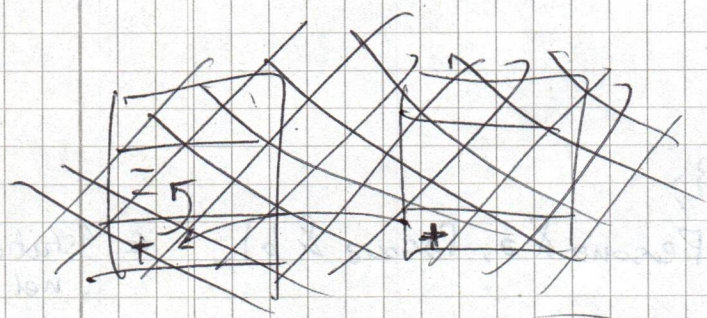
getter() ⟶ type getter();

setter(valori) ⟶ void setter(~~valori~~ variabili);

IF TERNARIO "?:"

```
Std::cout << ((1==2) ? "vero" : "falso");
```

- (1==2) → Cond.
- "vero" → se è vera
- "falso" → se è falsa



FRIEND ?

p.23 di Class 2: È UNA BUGIA

ERRORE ↑

OVERLOAD CON COSTRUTTORI (CLASSI)

```
Class Employee {
private:
    int eta;
    string name;

public:
    Employee() { }  ------------------>  ≡  Employee() = default;
    Employee(int a) : eta(a) { }
    ----
};
```