

# Lezione II: ripasso di C++ e STL

Laboratorio di Metodi Computazionali e Statistici (2023/2024)

Fabrizio Parodi

DIFI

# I/O

```
1 // File Vector.h
2 Vector operator*(double, Vector)
3 ;
4 class Vector{
5 public:
6     ...
7     void Print();
8     ...
9 private:
10     double m_v[3];
11 };

```

```
1 // File main.cpp
2 #include <iostream>
3 #include "Vector.h"
4 using namespace std;
5 int main(){
6     Vector a(1,1,1);
7     a.Print();
8     ...
9     return 0;
10 }

```

```
1 // File Vector.cpp
2 #include "Vector.h"
3 ...
4
5 void Vector::Print(){
6     cout<< "(" << m_v[0] << ", "
7         << m_v[1] << ", "
8         << m_v[2] << ")"
9         <<endl;
10
11     return;
12 }

```

# Ridefinizione dell'operatore <<

- Quando scriviamo

```
int i = 7;  
std::cout << i;
```

- In realtà stiamo chiamando la funzione

```
operator<<(std::cout, i)
```

e passando il risultato (per referenza) a std::cout

- Infatti la funzione è definita come

```
std::ostream& operator<<(std::ostream& o, int x)
```

- Esistono anche

```
std::ostream& operator<<(std::ostream& o, float x);  
std::ostream& operator<<(std::ostream& o, double x);  
std::ostream& operator<<(std::ostream& o, char x);  
std::ostream& operator<<(std::ostream& o, std::string s);
```

- Sta a noi definire

```
std::ostream& operator<<(std::ostream& o, Vector v);
```

# Ridefinizione dell'operatore <<

- Il fatto che la funzioni ritorni una referenza ad un oggetto `std::ostream` permette il concatenamento, che è in realtà una serie di chiamate ricorsive (nidificate) alla stessa funzione
  - Vediamo di capirlo assegnando un codice colore differente ad ogni chiamata della funzione

```
int i = 7;  
std::cout << " i = " << i << std::endl;
```

- In questo caso abbiamo tre chiamate. Partiamo da sx ricordando il prototipo della funzione  
`std::ostream& operator<<(std::ostream& o, const T &t)`

vediamo che quando scritto è equivalente a:

```
operator<<( operator<<( operator<<(std::cout, " i = "), i ), std::endl );
```

- Per usare nuovi tipi nella concatenazione è quindi necessario fare l'overloading della funzione << per il tipo in questione

# Ridefinizione dell'operatore <<

- Per la classe `Complesso` l'overloading può essere fatto, per esempio:

```
std::ostream &operator<<(std::ostream &stream, Vector v){  
    stream << "(" << v.X() << "," << v.Y() << "," << v.Z() << " )";  
    return stream;  
}
```

- Cioè si manda nello stream (che poi sarà `std::cout`)

- L'apertura di una parentesi tonda
- Il valore della X
- Una virgola
- "..."

- A questo punto il codice

```
Vector v(1,1,1);  
std::cout << "v = " << v << std::endl;
```

produrrà sul terminale

```
v = (1,1,1)
```

# Ridefinizione dell'operatore >>

- Per la classe `Complesso` l'overloading può essere fatto, per esempio:

```
std::istream &operator>>(std::istream &stream, Vector &v){  
    double x,y,z;  
    stream >> x >> y >> z;  
    v.X(x); v.Y(y); v.Z(z);  
    return stream;  
}
```

- Si noti come in questo caso `Vector` deve essere passato per referenza (deve essere modificato)

# Keyword *friend*

- Serve per permettere ad una funzione (o classe) esterna di accedere ai membri privati della classe
- Prima si dichiara la funzione (fuori dalla classe)
- Poi si dichiara la funzione *friend* della classe

```
1  void func();  
2  class myClass(){  
3      public:  
4          ...  
5      friend void func();  
6          ...  
7      private:  
8          ...  
9  }
```

la funzione *func* può quindi accedere ai membri privati della classe.

- Questa possibilità può essere molto utile nella ridefinizione di << e >> (per cui abbiamo finora usato l'accesso tramite metodi pubblici).

# Const e classi

const è una keyword importante del linguaggio C++. Garantisce maggiore modularità (promuovendo “l'accoppiamento debole”)

- const e argomenti di funzione

```
1 void print(int val)           // per valore, val
2                               // e' copiato (inefficiente)
3 void print(int& val)          // per ref. (piu' eff.)
4                               // ma print puo' cambiare val
5 void print(const int& val)    // per ref. (const),
6                               // print non puo' cambiare val
```

- La keyword const garantisce che l'operatore di assegnazione agisca correttamente:
  - Il compilatore può controllare l'assegnamento a const e, nel caso, mandare un messaggio di errore.
  - Tuttavia le classi possono cambiare i loro membri privati. Come controllare ?
  - Basta dire al compilatore quali metodi possono cambiare l'oggetto.




# Metodi const

Per default tutti i metodi possono cambiare un oggetto.

```
1  class Vector{
2  public:
3      ...
4      void Print();
5      ...
6  private:
7      double m_v[3];
8  };
9
10 void showVector(const Vector& v){
11     v.Print(); //Errore Print potrebbe cambiare v che e' passato
12               const !
13 }
14
15 int main(){
16     Vector v;
17     mostraVector(v);
18 }
```

# Metodi const

Soluzione: dichiarare che Print è una metodo che non può modificare i membri privati della classe.

```
1  class Vector{  
2  public:  
3      ...  
4      void Print() const;   
5      ...  
6  private:  
7      double m_v[3];  
8  };  
9  
10 void showVector(const Vector& v){  
11     v.Print(); // Ok  
12 }  
13  
14 int main(){  
15     Vector v;  
16     mostraVector(v);  
17 }
```

# Auto-riferimento

Come può un metodo di una classe fare riferimento all'oggetto stesso su cui sta agendo ?

- Per ogni istanza di una classe (oggetto) viene definito il puntatore all'oggetto stesso `this`
- Utile quando si vuol fare riferimento all'oggetto nella sua interezza

Immaginiamo di aver già definito l'operatore `+` (binario) e il `-` (unario).  
L'operatore binario `-` può essere definito come:

```
1 Vector Vector::operator-(const Vector& b){  
2     return Vector( (*this) + (-b) );  
3 }
```

- agisce su a,

# Namespace

- Ogni applicazione o programma complesso è spesso composto da più sorgenti; se la stessa variabile globale, funzione o classe è definita in diversi sorgenti si generano conflitti in compilazione.
- I **namespace** evitano tale conflitto fornendo un “contenitore” ad un gruppo di variabili, funzioni, classi (membri del namespace).

```
namespace first{  
    int var = 5;  
}  
namespace second{  
    double var = 3.1416;  
}
```

# Namespace

- Per accedere al singolo membro del namespace si usa l'operatore di risoluzione di ambito `::` (scope).

```
int main(){
    cout << first::var << endl;
    cout << second::var << endl;
    return 0;
}
```

- Alternativamente si può indicare una volta per tutte il namespace che si intende utilizzare evitando poi di specificarlo quando si utilizzano i singoli membri.

```
using namespace second;
int main(){
    cout << var << endl;
    return 0;
}
```

# Namespace

- Questo è esattamente quello che facciamo con le funzioni e classi della libreria standard quando scriviamo `using namespace std;`

```
#include <iostream>
using namespace std;
int main(){
    cout << "Ciao Mondo! " << endl;
    return 0;
}
```

- Alternativamente si può mettere esplicitamente l'operatore di scope

```
#include <iostream>
int main(){
    std::cout << "Ciao Mondo! " << std::endl;
    return 0;
}
```

# Direttiva #ifndef

- Capita spesso, in progetti complessi, di dover includere a cascata lo stesso include file .h a cascata.
- Per evitare inclusioni multiple (e quindi errori di compilazione) si usa il costrutto:

```
1 #ifndef _NOMEINCLUDE      se non def (inclusa già)
2 #define _NOMEINCLUDE      definisco (includo)
3     Type fun (type1 , type2 , ... ) ;
4 #endif
```

- Alla prima inclusione viene definita la flag \_NOMEINCLUDE e le dichiarazioni contenute nel seguito.
- Ogni successiva inclusione non ha effetto perchè \_NOMEINCLUDE è già definito
- Rispettate la convenzione secondo cui il NOMEINCLUDE è il nome del sorgente (e anche del include). Cioè pippo.cpp, pippo.h (e all'interno di pippo.h direttiva ifndef con \_PIPPPO).

# Polimorfismo



# Polimorfismo



# Polimorfismo

- La frase chiave è **“un’interfaccia più metodi”**
  - Una sola interfaccia (nome di funzione, metodo o classe) applicabile ad argomenti di diverso tipo.
- Nel polimorfismo **“compile-time”** è il compilatore che decide quale implementazione dell’interfaccia chiamare. Due tipologie:
  - Overloading di funzioni e metodi
  - Template: un solo algoritmo generico applicabile a diversi tipi
- Il polimorfismo **“run-time”** serve nei casi in cui il tipo di un oggetto non è noto al momento della compilazione ma solo durante l’esecuzione del programma
  - quindi la risoluzione della corretta implementazione viene fatta dinamicamente durante l’esecuzione
  - questo tipo di polimorfismo richiede classi virtuali, ereditarietà e allocazione dinamica di oggetti

# Overloading

- Overloading di funzioni e metodi.
  - Funzioni o metodi con lo stesso nome, ma diverso prototipo
  - Più esattamente, con diversa lista degli argomenti
- Il compilatore decide (compile-time) quale sia la funzione (o metodo) da utilizzare in una determinata circostanza in base alla lista degli argomenti. Dà errore se
  - Ci sono due funzioni con lo stesso nome e la stessa lista di argomenti
  - Se non esiste una versione della funzione con la lista corretta di argomenti

# Polimorfismo (con overloading)

```
1  int Max(int min, int max){
2      int tmax=max;
3      if (min>max) tmax=min;
4      return tmax;
5  }
6
7  float Max(float min, float max){
8      float tmax=max;
9      if (min>max) tmax=min;
10     return tmax;
11 }
12
13 double Max(double min, double max){
14     double tmax=max;
15     if (min>max) tmax=min;
16     return tmax;
17 }
```

# Cosa sono i template ?

- Se un algoritmo è indipendente dal tipo degli argomenti è possibile definire una funzione in modo parametrizzato (template), dove il tipo (o i tipi) non è definito in modo esplicito ma è un parametro.
- Questa funzione templata potrà essere utilizzata (specializzata) senza alcuna modifica di codice per qualsiasi nuovo tipo (classe).

# Esempio (con template)

```
1  template <class T>                                tipo viene sostituito dal tipo utilizzato
2  T Max(T min, T max){
3      T tmax=max;
4      if (min>max) tmax=min;
5      return tmax;
6  }
7
8  double a=5.,b=6.;
9  cout << Max(a,b) << endl;    // chiama Max(double, double)
10
11 int i=5,j=6;
12 cout << Max(i,j) << endl;    // chiama Max(int, int)
```

La keyword `class` non ha qui il significato usuale: indica che `T` è il nome di un tipo (anche nativo), e non necessariamente di una classe.

# Classi template

- È anche possibile definire classi template
  - In questo caso il tipo di uno o più dati membro diventa un parametro

```
1 template <class T>
2 class complex{
3     public:
4         ...
5     private
6         T im, re;
7 }
8
9 complex          myComplex;           // Errore di compilazione
10 complex<int>      myIntComplex;       // complesso di interi
11 complex<double>   myDoubleComplex;    // complesso di double
```

# Template: osservazioni

- La definizione di una funzione template non è ancora codice, ma lo diventerà una volta che essa viene usata.
- Il fatto che il compilatore generi codice concreto solo una volta che una funzione è usata ha come conseguenza che una template function non può essere mai raccolta in una libreria precompilata.
- Un template va inteso come una sorta di dichiarazione e fornito in un header file (che però deve contenere dichiarazione e definizione delle funzioni).



# STL: Standard Template Library

<http://www.sgi.com/tech/stl>

La stessa Libreria Standard del C++ mette a disposizione strutture precostituite di classi template. In particolare, La Standard Template Library fornisce:

- **container** che rappresentano le strutture dati di base;
- **iteratori** che generalizzano il concetto di puntatore C/C++;
- **algoritmi generici** che operano sui container attraverso iteratori

Combinazioni di algoritmi, container e iteratori: realizzazione di componenti di alto livello garantendo efficienza e ri-utilizzabilità.

# Contenitori sequenziali

- **vector**: il contenitore più completo; memorizza un array monodimensionale, ai cui elementi può accedere in modo "random", tramite iteratori ad accesso casuale e indici; può modificare le sue dimensioni, espandendosi in base alle necessità.
- **list**: rispetto a vector manca dell'accesso tramite indice e di varie operazioni sugli iteratori, che non sono ad accesso casuale ma bidirezionali; è più efficiente di vector nelle operazioni di inserimento e cancellazione di elementi.
- **deque**: misto tra vector e list: ha accesso random ed è più efficiente di vector per inserimenti in testa ed in coda. Attenzione: non è però garantito che gli elementi siano contigui in memoria

```
1 vector<double> v;  
2 ...  
3 double *p;  
4 p = v;  
5 cout << p[1] << endl;    // Se esiste e' sicuramente la seconda  
6                           // componente del vector v.  
7                           // Non vero in generale per deque
```

# Vector

- `vector` è la generalizzazione della STL dell'array C (e gestisce la memoria autonomamente)
- Un oggetto memorizzato in un `vector` dovrà possedere almeno un costruttore e gli operatori di `<` e `==`
- Alcuni esempi:

```
1 #include <vector>
2 vector<int> iv           // vett. int di lunghezza 0
3 vector<float> v(5)       // vett. float da 5 elem.
4 vector<float> v(5,0)     // vett. float da 5 elem. = 0
```

# Vector

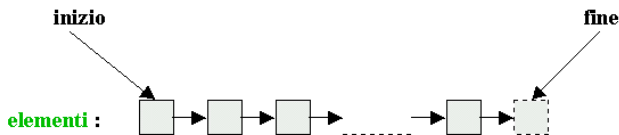
- È definito l'operatore di indicizzazione `[]` che opera in modo simile agli array
- Esistono diverse funzioni membro, tra le quali
  - `int size()`  
restituisce la dimensione corrente del vettore
  - `void push_back(const T& x)`  
aggiunge un valore alla fine del vettore (espandendone dinamicamente le dimensioni se necessario)
  - `void clear()`  
cancella tutti gli elementi
  - `T& at(int i)`  
come `[]` ma controlla che l'indice sia valido.

# Vector: esempio

```
1 #include <vector>
2 vector<int> v;
3 v.push_back(3);           // inserisce 3
4 v.push_back(4);           // inserisce 4
5 cout << v.size() << endl; // stampa 2
6 cout << v[0] << " " << v[1] << endl; // stampa 3 4
```

# Iteratori

Sono una generalizzazione dei puntatori C++ in grado di lavorare in maniera uniforme su strutture dati di tipo diverso.



Un iteratore "punta" a un elemento e fornisce un'operazione per far sì che l'iteratore stesso possa puntare all'elemento successivo della sequenza. La fine di una sequenza corrisponde a un iteratore che "punta" all'ipotetico elemento che segue immediatamente l'ultimo elemento della sequenza (non esiste un iteratore NULL, come nei normali puntatori).

# Iteratori

## Operazioni fondamentali

- accesso all'elemento puntato, tramite dereferenziazione `*` o `->`
- passaggio all'elemento successivo (operatore `++`).
- test di uguaglianza `==` o disuguaglianza `!=`

## Dichiarazione:

```
container<T>::iterator it;
```

## Metodi principali:

- `v.begin()` ritorna un iteratore che punta al primo elemento del vettore
- `v.end()` ritorna un iteratore che punta all'ipotetico elemento che segue l'ultimo (e pertanto non si deve mai dereferenziale).

# Iteratori: esempio

```
1  vector<double> v;  
2  v.push_back(5.0);  
3  v.push_back(2.0);  
4  v.push_back(7.0);  
5  vector<double>::iterator it;  
6  vector<double>::iterator end=v.end();  
7  for (it=v.begin(); it!=end; it++)  
8      cout << *it << endl;
```



# Iteratori: esempio (C++11)

- `auto` è una nuova parola *keyword* del linguaggio che consente di dichiarare oggetti senza doverne specificare esplicitamente il tipo qualora la dichiarazione dell'oggetto includa già un'inizializzazione.

```
1 vector<double> v;  
2 v.push_back(5.0);  
3 v.push_back(2.0);  
4 v.push_back(7.0);  
5 for (auto it=v.begin(); it!=v.end(); it++)  
6     cout << *it << endl;
```

# Iteratori: esempio (C++11)

- **range based for loop** opera su strutture  $x$  che supportino il concetto di iterazione, ovvero:
  - abbiano le funzioni membro  $x.begin()$  e  $x.end()$  , oppure,
  - abbiano le funzioni non-membro  $begin(x)$  e  $end(x)$  , oppure,
  - per le quali esistono le specializzazioni di  $std::begin(x)$  e  $std::end(x)$ .

```
1  vector<double> v;  
2  v.push_back(5.0);  
3  v.push_back(2.0);  
4  v.push_back(7.0);  
5  
6  for (double x: v)  
7      cout << x << endl;  
8  
9  // oppure  
10 for (auto x: v)  
11     cout << x << endl;
```

# Iteratori

Metodi di `vector` che prendono gli iteratori come parametri<sup>1</sup>

```
1  v.insert(v.begin(),5);  
2  // inserisce in cima a v 5  
3  v.erase(v.begin());  
4  // cancella il primo elemento  
5  v.erase(v.begin(),v.end());  
6  // cancella tutti gli elementi di v
```

<sup>1</sup>Per la lista completa consultare ad esempio

<http://www.cplusplus.com/reference/stl/vector/>

# Contenitori associativi

- **map**: è il più importante dei contenitori associativi; memorizza una sequenza di coppie (chiave e valore mappato, entrambi parametri di map) e fornisce un'accesso rapido a ogni elemento tramite la sua chiave (ogni chiave deve essere unica all'interno di un map); mantiene i propri elementi in ordine crescente di chiave.

L'operazione caratteristica su di esso è l'accesso tramite chiave (chiamiamo m un oggetto di map):

```
valore mappato = m[chiave]    o  
m[chiave] = valore mappato
```

che funziona sia in estrazione che in inserimento; in ogni caso cerca l'elemento con quella chiave: se lo trova, estrae (o inserisce) il valore mappato.

- **set**: è un contenitore associativo analogo a map, con la differenza che possiede solo la chiave (e quindi ha un solo parametro); in pratica è una sequenza ordinata di valori unici e crescenti.
- **multimap**, **multiset**: analoghi con la differenza che la chiave può essere duplicata.

# Map

- La classe `map` definisce un contenitore associativo in cui delle chiavi (uniche) sono associate a dei valori
- Una mappa è quindi un insieme di coppie chiave/valore
- È ordinata, quindi per memorizzare un tipo di dato in una mappa devono essere definiti (oltre al costruttore) almeno gli operatori `<` e `==`.
- Dichiarazione:

```
map<Key,T> nome_mappa;
```

- `first`  
è il membro che contiene la chiave;
- `second`  
è il membro che contiene il valore;

# Map: esempio

```
1 #include <map>
2 map<char, int> m;
3 m['b'] = 5;
4 m['a'] = 7;
5 map<char, int>::iterator it = m.begin();
6 map<char, int>::iterator end = m.end();
7 while(it != end){
8     cout << "coppia: " << it->first << " "
9         << it->second << endl;
10    it++;
11 }
12 // stampa:
13 // coppia: a 7
14 // coppia: b 5
```

N.B. Gli elementi sono ordinati secondo la chiave

# Map: esempio (C++11)

```
1 #include <map>
2 map<char, int> m;
3 m['b'] = 5;
4 m['a'] = 7;
5 for (auto el: m)
6     cout << "coppia: " << el.first << " "
7         << el.second << endl;
8 }
```

# Set

Tipo di contenitore associativo in cui la chiave ed il valore coincidono

```
1 #include <set>
2 set<double> x;
3 x.insert(5);
4 x.insert(4);
5
6 for (auto comp : x)
7     cout << comp << endl;
8
9 // stampa '
10 // 4
11 // 5
```



# Algoritmi

La STL mette a disposizione una sessantina di funzioni template, dette "algoritmi" e definite nell'header-file `<algorithm>`.

Gli algoritmi operano sui contenitori, o meglio, su sequenze di dati. Fra gli argomenti di ingresso di un algoritmo è sempre presente almeno una coppia di iteratori (di tipo parametrizzato) che definiscono e delimitano una sequenza: il primo iteratore punta al primo elemento della sequenza, il secondo iteratore punta alla posizione che segue immediatamente l'ultimo elemento.

Nella chiamata di un algoritmo gli argomenti che esprimono i due iteratori devono essere dello stesso tipo. A parte questa limitazione (peraltro ovvia), gli algoritmi sono perfettamente generici, nel senso che possono operare su qualsiasi tipo di contenitore (e su qualsiasi tipo degli elementi), purché provvisto di iteratori.

# Algoritmi

Alcuni esempi:

- `void sort(iterator begin, iterator end)`  
ordina gli elementi in ordine ascendente secondo il `<` definito per la classe `T`
- `void sort(iterator begin, iterator end, bool cmp(const T&, const T&))`  
ordina gli elementi del vettore in ordine ascendente utilizzando il risultato di `cmp` invece di `<`.
- `iterator find(iterator start, iterator end, const T&val)`  
cerca il valore `val` tra `start` e `end`. Se lo trova ne ritorna l'iteratore altrimenti ritorna un iteratore a `end()`

# Funzione template (in C++)

In C++ è possibile utilizzare una classe della Standard Template Library

- Dichiarazione:

```
1  std::function <Return_type(parameters)>  
    nome_funzione_generica;
```

- Esempio:

```
1  #include <functional>  
2  double square(double x) {  
3      return x*x ;  
4  }  
5  int main() {  
6      std::function <double(double)> generic_f;  
7      generic_f = square ;  
8      cout << square(5.0) << endl ;      // Chiamata diretta  
9      cout << generic_f(5.0) << endl ;    // Chiamata via classe  
        template  
10 }
```

# Esempi

- Ordinamento di un vettore di numeri
- Elenco telefonico (associazione nome-numero)
- Elenco studenti ordinati per anno di nascita

# Ordinamento di un vettore

```
1 int main(){
2     vector<double> v;
3     v.push_back(1.0);
4     v.push_back(3.0);
5     v.push_back(2.0);
6     v.push_back(10.0);
7
8     sort(v.begin(), v.end());
9
10    auto it=v.begin();
11    while(it!=v.end()){
12        cout << *it++ << endl;
13    }
14
15 }
```

# Ricerca in un mini-elenco telefonico

```
1 #include <map>
2 int main(){
3     map<string ,int> elenco ;
4     elenco["Rossi"]    = 340352015;
5     elenco["Verdi"]    = 335289751;
6     elenco["Bianchi"]  = 318103456;
7
8     auto it=elenco.begin();
9     while(it!=elenco.end()){
10         cout << it->first << " " << it->second << endl;
11         it++;
12     }
13
14     it=elenco.find("Rossi");
15     if (it!=elenco.end())
16         cout << it->second << endl;
17     else
18         cout << "Not found" << endl;
19
20 }
```

# Elenco studenti ordinati per anno di nascita

Supponiamo di voler ordinare per anno di nascita un'elenco di studenti.

- Creiamo intanto una classe contenente cognome ed anno di nascita per ogni studente.
- Definiamo il  $<$  di quella classe in modo che lavori sull'anno
- Creiamo un vettore di oggetti
- Utilizziamo sort per ordinarli

# Elenco studenti ordinati per anno di nascita

Creiamo intanto una classe contenente cognome ed anno di nascita per ogni studente.

```
1 class stud{
2 public :
3     stud(string nome="",int anno=0):m_nome(nome),m_anno(anno){}
4     bool operator<(const stud& s) const {return m_anno< s.m_anno;}
5     int    anno() const {return m_anno;}
6     string nome() const {return m_nome;}
7 private :
8     string m_nome;
9     int    m_anno;
10 };
```



# Elenco studenti ordinati per anno di nascita

```
1 int main(){
2
3     vector<stud> elenco;
4     stud p1("Rossi", 1989);
5     stud p2("Verdi", 1990);
6     stud p3("Bianchi", 1988);
7     elenco.push_back(p1);
8     elenco.push_back(p2);
9     elenco.push_back(p3);
10
11     sort (elenco.begin(), elenco.end());
12
13     for (auto s:elenco)
14         cout << s.nome() << " " << s.anno() << endl;
15
16     return 0;
17
18 }
```

## Elenco studenti ordinati per anno di nascita

Analogo ma con funzione di confronto esterna: non è necessario definire il minore nella classe stud (utile soprattutto se non si è il “possessore” della classe).

```

1 bool comp(const stud& a, const stud& b){
2     if (a.anno()<b.anno())
3         return true;
4     else
5         return false;
6 }
7
8 int main(){
9     vector<stud> el;
10    el.push_back(stud("Rossi",1989));    // simile a prima... solo
11    el.push_back(stud("Verdi",1990));    // p1, p2... non sono creati
12    el.push_back(stud("Bianchi",1988));  // ma vengono creati al volo
13                                         // gli stud da mettere in el
14    sort (el.begin(),el.end (),comp) ;
15
16    for (auto s:el){
17        cout << s.nome() << " " << s.anno() << endl;
18    }
19
20    return 0;
21

```

## Esercizio riassuntivo

Si parte dalla classe `Vector.h`, `Vector.cpp`

- Estensione delle funzionalità: modulo e versore (entrambi sfruttando *this*) e opzionalmente prodotto scalare e vettore (non saranno sfruttate nell'esercitazione ma sono in generale utili per una classe `Vector` completa)
- Creazione di un programma di test che:
  - Lettura da file di un vector di `Vector` (da file `punti.dat`)
  - Ordinamento in base al modulo (sort con funzione di confronto esterna alla classe)
  - Calcolo del vettore somma (con range-based-for-loop)

Per partire da quanto fatto a lezione date il comando

```
git clone https://github.com/fabrizio-parodi/LabMCS-Es0.git Es0
```