ROMA TRE
UNIVERSITÀ DEGLI STUDI

UNIVERSITA' DEGLI STUDI ROMA TRE

DIPARTIMENTO DI FISICA "E. AMALDI"

Laboratorio di
Programmazione e Calcolo

6 crediti

a cura di

Severino Bussino

Anno Accademico 2021-22

0) Struttura del Corso

1) Trattamento dell'informazione
Elementi di Architettura di un Computer

Verra' trattata in una delle prossime lezioni

2) Sistemi operativi

3) Introduzione alla
Programmazione ad oggetti (OO)

4) Simulazione del Sistema Solare

5) Introduzione al linguaggio C/C++

6) Elementi di linguaggio C/C++

A
1 - istruzioni e operatori booleani
2 - iterazioni (`for`, `while`, `do … while`)

B
- istruzioni di selezione (`if`, `switch`, `else`)

C
- funzioni predefinite. La classe `math`.

7) Puntatori

8) Vettori (Array)

9) Vettori e Puntatori

10) Classe SistemaSolare (prima parte)

11) Gestione dinamica della memoria

12) Classe SistemaSolare

13) Programma di Simulazione (main)

14) Ereditarieta'

- Concetto
- Funzione
- Realizzazione in C++
- Esempi

15) Classe Sonda

16) Output su file

17) Polimorfismo

- Concetto
- Funzione
- Realizzazione in C++
- Esempi

18) “per referenza” e “per valore”

19) Richiami sulle funzioni

20) Cenni alle classi Template

21) La libreria STL
(Standard Template Library)

22) L'algebra di una classe (cenni)

23) L'algebra di una classe (complementi)

Valutazione del Corso (I)

ezioni | Calendario delle Esercitazioni | Materiale Didattico | Prove di Valutazione

You are here: Home › Prove di Valutazione › Test a risposte multiple

Test a risposte multiple

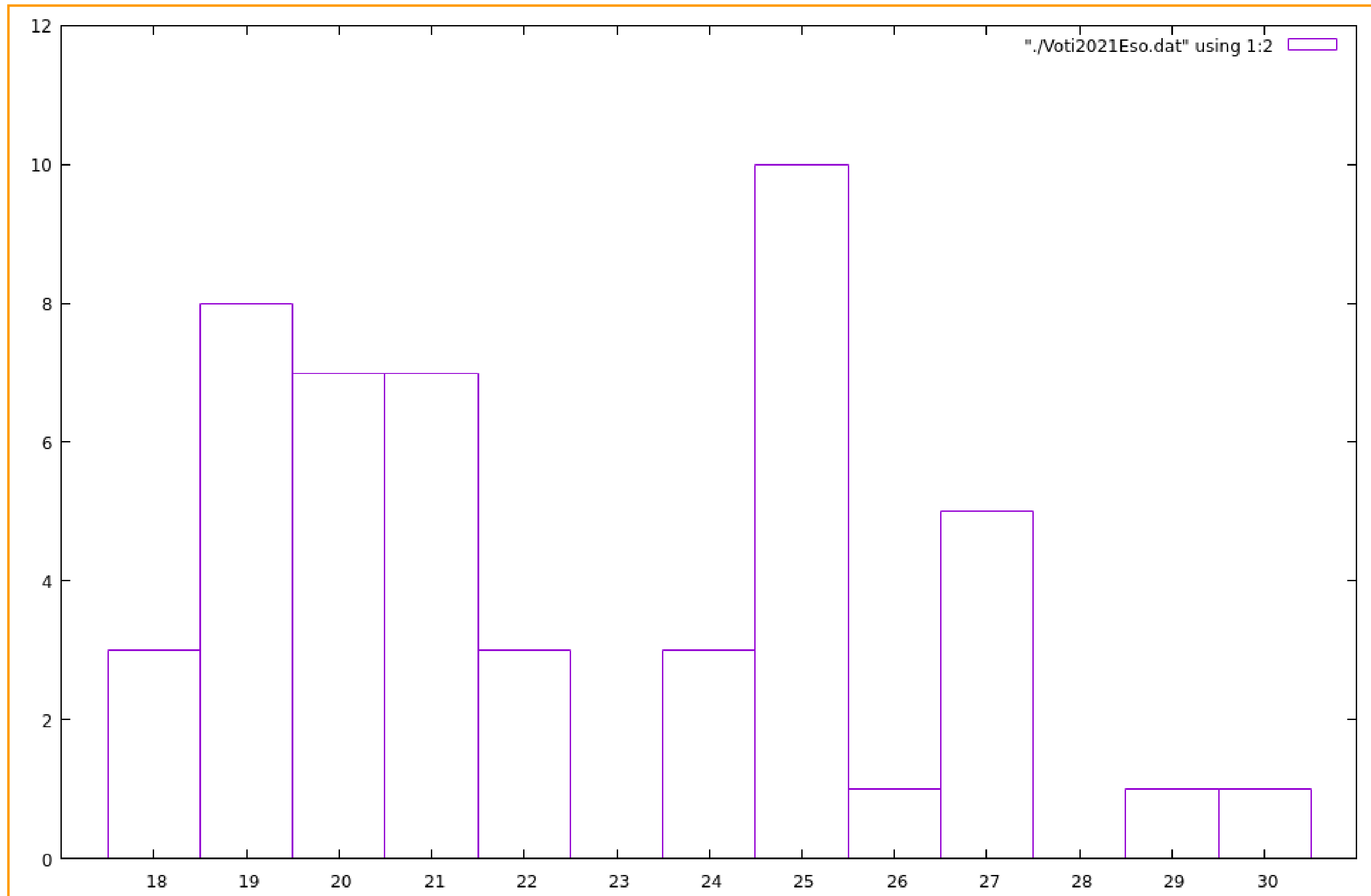
Un test di esonero a risposte multiple si svolgera'

Venerdi' 12 novembre 2021 in aula M3

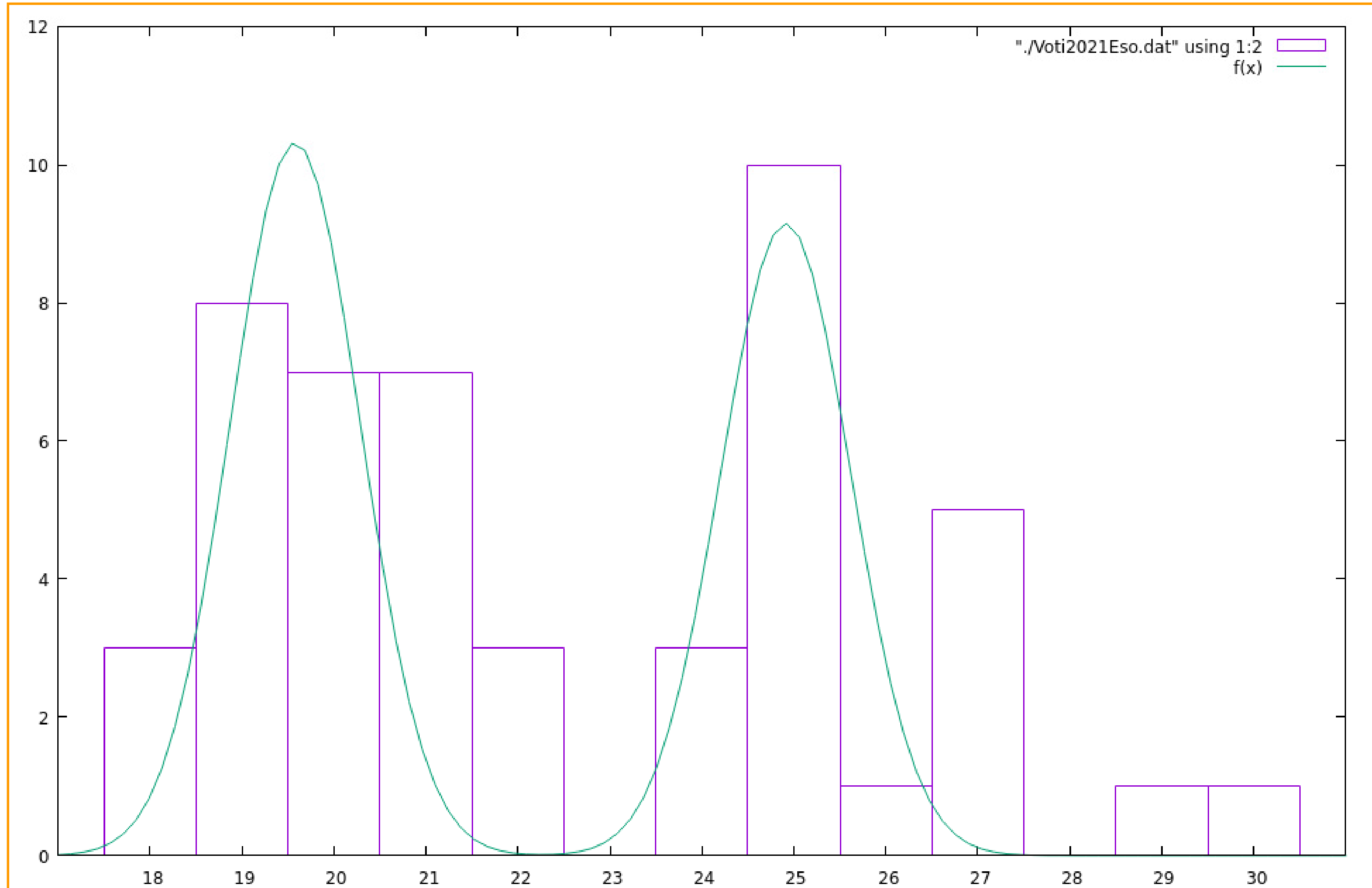
dalle 11:00 alle 13:00

(Largo San Leonardo Murialdo, 1)

Valutazione del Corso (II)



Valutazione del Corso (III)



Le lezioni in aula si concludono il 3 dicembre

Venerdi' 26 novembre NON ci sara' lezione

Le Esercitazioni di Laboratorio proseguono senza variazioni con l'orario gia' comunicato (e pubblicato sul sito)

16 nov - 17 nov - 18 nov Classe Shape (Ereditarieta')

23 nov - 24 nov - 25 nov Classe Shape (sec. parte se necessario ...)

30 nov - 1 dic - 2 dic Classe Sonda + container STL per i pianeti

14 - 15 - 16 dicembre Simulazione Prova Individuale

11 - 12 - 13 gennaio Prova Individuale

20) Cenni alle classi Template (Programmazione Generica)

Premessa (1)

Relazioni tra Concetti ed Ereditarieta'

«A concept does not exist in isolation. It coexists with related concepts and derives much of its power from relationships with related concepts…» (*B. Stroustrup*)

Indipendenza tra Concetti e Programmazione Generica

«Independent concepts should be independently represented and should be combined only when needed. Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies. Either way, you get a less flexible set of components out of which to compose systems…» (*B. Stroustrup*)

settima lezione

cenni oggi

Premessa (2)

L'implementazione di un metodo assume molte forme (poli-morfismo) ed e' possibile scegliere "automaticamente" la "forma" giusta

Polimorfismo a *Run Time*

*La forma "giusta" viene scelta durante l'**esecuzione** del programma*

Polimorfismo a *Compilation Time*

*La forma "giusta" viene scelta durante la **compilazione** del programma*

Programmazione Generica

“Indice”

1. Il problema (cosa vorremmo fare)
   - Un esempio: una matrice di …
   - Calcolo del minimo tra due oggetti (per i quali e’ definito un ordinamento)
2. La soluzione: le classi *Template*
   - Un paragone (ingenuo)
   - Uso
   - Sintassi
3. Introduzione alla libreria STL
   - Uso
   - Riferimenti
   - Contenitori, Algoritmi, *Utilities*
4. I *container*
   - Uso di vector
   - Iteratori
   - Funzioni

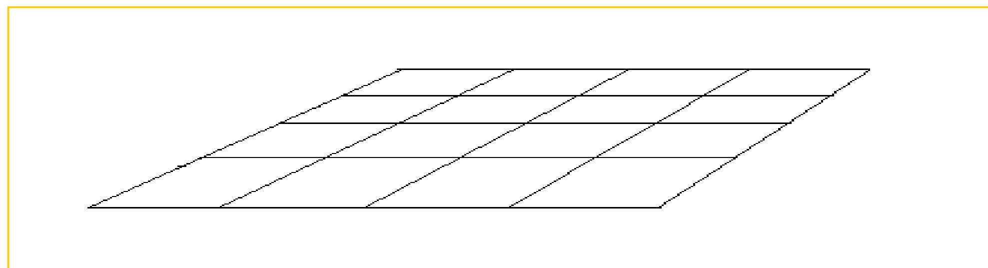
Cenni alle classi Template
(Programmazione Generica)

La libreria STL
(Standard Template Library)

Il Problema (cosa vorremmo (poter) fare) (1)

Un esempio!

Voglio contare le gocce di pioggia che cadono sulle mattonelle di un pavimento

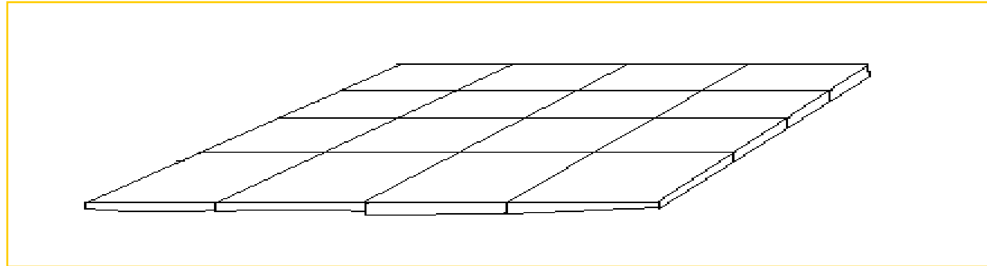


```
class Myfloor {
   protected :
        int element[4][4] ;
   public:
     // costruttori
     // distruttore
     // metodi di tipo set e di tipo get
     // altri metodi (incremento) e operatori
     ..................................
 } ;
```

Il Problema (cosa vorremmo (poter) fare) (2)

Un altro esempio!

Voglio studiare la distribuzione di massa in una lastra non omogenea

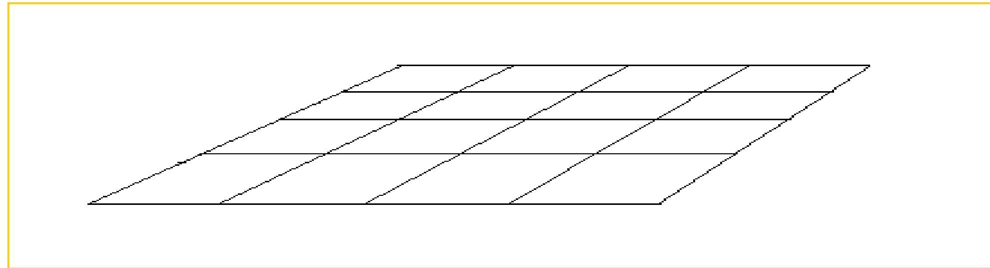


```
class Myplate {
   protected :
        double element[4][4] ;
   public:
     // costruttori
     // distruttore
     // metodi di tipo set e di tipo get
     // altri metodi e operatori
     .....................................
 } ;
```

Il Problema (cosa vorremmo (poter) fare) (3)

Un altro esempio!

Voglio studiare il campo elettrico su un reticolo



```
class Myfield {
   protected :
        ThreeVector element[4][4] ;
   public:
     // costruttori
     // distruttore
     // metodi di tipo set e di tipo get
     // altri metodi e operatori
     ………………………………
 } ;
```

Il Problema!

Stiamo scrivendo molte linee di codice praticamente identiche

Eppure non possiamo ri-utilizzare il codice attraverso il meccanismo dell'Ereditarieta'!

Un altro esempio

Vogliamo calcolare il minimo tra due oggetti
(per i quali sia definito un ordinamento)

Definiamo una funzione minimo

```
double min(const double & a1, const double & a2) {
   if(a1<=a2) { return a1; } else { return a2;}
}
```

```
int min(const int & a1, const int & a2) {
   if(a1<=a2) { return a1; } else { return a2;}
}
```

```
TwoVect min(const TwoVect & a1, const TwoVect & a2) {
   if(a1<=a2) { return a1; } else { return a2} ;
}
```

E il ri-utilizzo del codice?

La soluzione: le classi *Template* (1)

1. Un paragone (ingenuo)

x	$f(x) = x^2$
1	1
2	4
2.5	6.25
3	9
....	....

La soluzione: le classi *Template* (2)

2. Uso

Posso usare un parametro T che parametrizzi il tipo di classe da utilizzare

La programmazione e' generica poiche' non dipende esplicitamente dal tipo di classe specifica:
T indica **genericamente** una classe

La soluzione: le classi *Template* (3)

3. Sintassi - La dichiarazione della classe (header file - `.h`)

```
template <class T> class Myclass {
   ………………………………….
    // T si puo’ usare come
    // una classe qualsiasi
   ………………………………….
};
```

- La dichiarazione dei metodi avviene come se T fosse una qualsiasi altra classe

La soluzione: le classi *Template* (4)

3. Sintassi - L'implementazione della classe
(implementation file - `.icc`)

```
template <class T>
  tipo Myclass<T>::nomemetodo(…) { ………….. }
```

- L'implementazione dei metodi avviene come se T fosse una qualsiasi altra classe
- L'*implementation file* di una classe template si chiama `.icc` e deve essere incluso alla fine del corrispondente *declaration file* `.h`
- Il file `.icc` non deve essere compilato

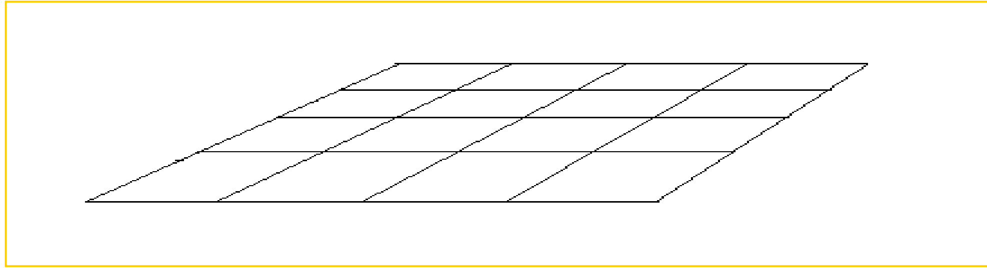
Attenzione!

Se non si vuole usare il file di tipo `.icc`, e' necessario far precedere l'implementazione del metodo da `export`

```
export template <class T>
  tipo Myclass<T>::nomemetodo(…) {…}
```

Non piu' implementata!

L'esempio precedente!



```
template <class T> class Mynet {
   protected :
        T element[4][4] ;
   public:
     // costruttori
     // distruttore
     // metodi di tipo set e di tipo get
     // altri metodi e operatori
     ...................................
 } ;
```

Riesco a ri-utilizzare il codice!

```
Myfloor
```

```
Mynet<int>
```

```
Myplate
```

```
Mynet<double>
```

```
Myfield
```

```
Mynet<ThreeVector>
```

Un esempio di programma `main()`

```
#include <Mynet.h>
int main()  {
 ………………………………
  Mynet<int>  mi;
  Mynet<double> mdb;
 ………………………………
return 0;
}
```

L'esempio precedente: il *declaration file* `Mynet.h`

```
#ifndef MYNET_H
#define MYNET_H

#include <iostream>
using namespace std ;

template <class T> class Mynet {

   protected :
     T element[4][4] ;

   public:
     //costruttori e distruttore
     Mynet() { } ;

     ~Mynet() ;

     // metodi di tipo set e di tipo get
     void set_element( int &i, int & j, const T & value ) ;
     T get_element( int &i, int & j ) ;

  } ;

#include "Mynet.icc"

#endif
```

L'esempio precedente: l' *implementation file* `Mynet.icc`

```
// file .icc (include implementation file) della classe
// template Mynet<T>

   //costruttori (e' implementato nel .h)

   //distruttore
     template <class T> Mynet<T>::~Mynet() {

       cout << " .. sto distruggendo un oggetto della "
                              << "classe template Mynet " << endl ;

     }


   //metodi di tipo set e di tipo get
     template <class T> void Mynet<T>::set_element
                           ( int &i, int &j, const T & value ) {

       element[i][j] = value;

     }


     template <class T> T Mynet<T>::get_element( int &i, int &j ){

       return  element[i][j] ;

     }
```

L'esempio precedente: un programma `main()` (1)

```
#include "Mynet.h"
#include <iostream>
using namespace std;

int main() {

  Mynet<int>    mi ;
  Mynet<double> md ;


 for (int i=0; i<4; i++) {
  for (int j=0; j<4; j++) {

      mi.set_element(i, j,        10*i+j    )      ;
      md.set_element(i, j,  10*i+ (double) j/10. ) ;
  }
 }


                 // continua
```

L'esempio precedente: un programma `main()` (2)

```
cout << endl << " Ecco Mynet<int> " << endl ;

for (int i=0; i<4; i++) {
 for (int j=0; j<4; j++) {

    cout << " " << mi.get_element( i, j ) ;

 }
 cout << endl ;
}


cout << endl << " Ecco Mynet<double> " << endl ;

for (int i=0; i<4; i++) {
 for (int j=0; j<4; j++) {

    cout << " " << md.get_element( i, j ) ;

 }
 cout << endl ;
}

cout << endl ;
return 1;

}
```

L'esempio precedente: un programma `main()` (3)

```
nbseve(~/templates)>./prvMynet

 Ecco Mynet<int>
 0 1 2 3
 10 11 12 13
 20 21 22 23
 30 31 32 33

 Ecco Mynet<double>
 0 0.1 0.2 0.3
 10 10.1 10.2 10.3
 20 20.1 20.2 20.3
 30 30.1 30.2 30.3

 .. sto distruggendo un oggetto della classe template Mynet
 .. sto distruggendo un oggetto della classe template Mynet
nbseve(~/templates)>
```

Se lo ritengo opportuno, posso poi usare anche l'Ereditarieta', implementando solo i costruttori ed eventuali metodi specifici

```
class Myfloor : public Mynet<int> {
    public:
     // costruttori …
     ~Myfloor() {}; // distruttore
} ;
```

```
class Myplate : public Mynet<double> {
    public:
     // costruttori …
     ~Myplate() {}; // distruttore
} ;
```

```
#include <TwoVector.h>
class Myfield : public Mynet<TwoVector> {
    public:
     // costruttori …
     ~Myplate() {}; // distruttore
} ;
```

…piu' chiaramente…

Myfloor.h

```
#ifndef MYFLOOR_H
#define MYFLOOR_H

#include "Mynet.h"

class Myfloor : public Mynet<int> {

    public:

    // costruttori
    Myfloor() : Mynet<int>() {} ;
    //Myfloor() {} ;

    // distruttore
    ~Myfloor() {} ;


} ;
```

Myfloor.cc

```
#include "Myfloor.h"
```

Nel `main()` precedente si puo' sostituire

```
Mynet<int>   mi ;
```

con

```
Myfloor      mi ;
```

Il calcolo del minimo! (1)

La funzione :

Mymin.icc

```
template <class T> T & min(const T & a1, const T & a2) {
   if(a1<=a2) { return a1; } else { return a2;}
}
```

Il calcolo del minimo! (2)

E il suo uso:

```
nbseve(~/templates)>./provamin
  2   4   bussino
nbseve(~/templates)>
```

```
#include <string>
#include <iostream>
using namespace std;
int main()  {
  float a = min(3.,2.);  //float
  int i   = min(4,6) ;   // int
  string s = min("severino" , "bussino " ); //stringhe
  cout << "  " << a << "  " << i << "  " << s << endl ;
return 1;
}
#include “Mymin.icc”
```

Un commento

1. E' frequente e semplice utilizzare le classi Template scritte da altri (un esempio tra poco)

2. E' piu' delicato scrivere una classe con uno o piu'Template (ma questo va oltre il livello di questo corso)

E' bene sapere

- che i Template esistono
- come si usano

L'istruzione `typedef`

```
typedef type nicktype ;
```

`type` e' un tipo noto al compilatore (es. `float`, `int`, `CorpoCeleste`, ecc.)

`nicktype` e' un soprannome che si attribuisce al tipo `type`

Molto semplice e molto utile, soprattutto nel caso di template.
Ad esempio:

```
typedef Mynet<int> Myint ;

typedef Pair<string*, int> VotiLC ;

...........

VotiLC * AA2011[30] ;
```

23) La libreria STL (Standard Template Library)

Introduzione alla libreria STL (1)

http://www.cplusplus.com/reference/

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

1. Riferimenti

C++ Standard Library: Standard Template Library (STL)

Containers library:

bitset	Bitset (class template)
deque	Double ended queue (class template)
list	List (class template)
map	Map (class template)
multimap	Multiple-key map (class template)
multiset	Multiple-key set (class template)
priority_queue	Priority queue (class template)
queue	FIFO queue (class template)
set	Set (class template)
stack	LIFO stack (class template)
vector	Vector (class template)

Iterators library:

iterator	Iterator definitions (header)

Algorithms library:

STL Algorithms	Standard Template Library: Algorithms (library)

Numeric library:

complex	Complex numbers library (header)
valarray	Library for arrays of numeric values (header)
numeric	Generalized numeric operations (header)

Introduzione alla libreria STL (2)

2. Uso

   - Contenitori
   - Iteratori
   - Algoritmi

3. I *container*

   Strutture dati per memorizzare oggetti
   (e puntatori ad oggetti)

   Per accedere ad essi (iteratori)

   Per eseguire operazioni su di essi
   (*find*, *sort*, *copy*, *merge…*)

Introduzione alla libreria STL (3): i *container*

4. Esempi di contenitori

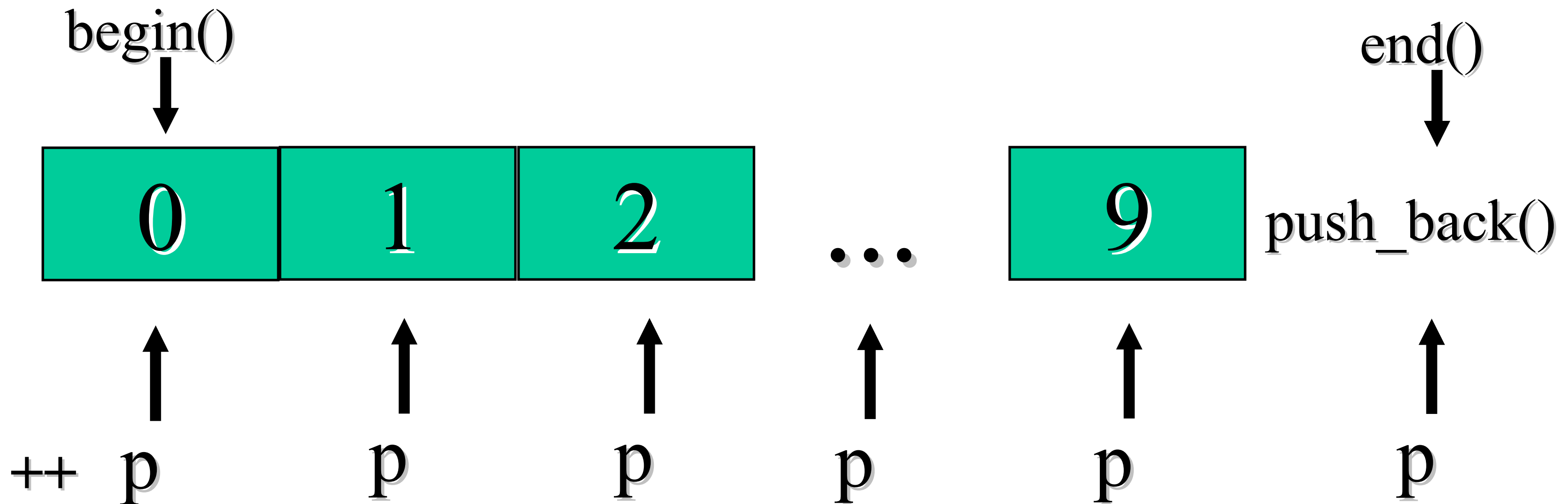
   vector - `vector<T>`

   list - `list<T>`

   map - `map<key,T>`

vector: 1 - Come funziona

vector e' un array contiguo di oggetti



`push_back(value)` e' un metodo della classe vector per:

- aggiungere alla fine del vettore un nuovo elemento
- assegnare al nuovo elemento il valore `value`

vector: 2 - gli iteratori

Gli iteratori permettono di scorrere il container e si comportano come i puntatori

Se v e' un container di tipo vector

`v.begin()` - e' un iteratore
punta all'inizio del container

`v.end()` - e' un iteratore
punta alla prima locazione **dopo** la fine

```
vector<T>::const_iterator p ;
```

Si puo' usare il typedef

```
typedef vector<T>::const_iterator viter ;
```

vector: 3 - Come si usa

```
nbacer(~/lez_9)>./prvector
0
10
0 1 2 3 4 5 6 7 8 9
nbacer(~/lez_9)>
```

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;

int main() {

  vector<int> v;                   // create an empty vector of integers

  cout << v.size() << endl;        // print the size of v: zero

  for (int i=0; i!=10; i++) {      // a loop from 0 to 9
     v.push_back(i);               // add an integer to v from the back
  }

  cout << v.size() << endl;        // print the size of v: 10

  // create an constant iterator for a vector of integers:
  // p behaves at all effects as a "const int *"
  vector<int>::const_iterator p;

  //  begin() points to the first element and  end() to the last+1
  // (compare with the previous "for" loop)

  for (p=v.begin(); p!=v.end(); ++p) {
      cout << (*p) << " ";
  }

  cout << endl;
  return 0;
}
```

Vector of abstract class Shape*

```
#include "Shape.h"
#include "Cerchio.h"
#include "Rettangolo.h"
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;

int main() {

  vector<Shape *> vs;           // a vector of pointers to Shapes
  for (int i=0; i!=10; i++) {
    vs.push_back(new Cerchio(i));         // add a new pointer to a new Circle
    vs.push_back(new Rettangolo(i, i+1)); // add a new pointer to a new Rectangle
  }

  // iterator: essentially a pointer to Shape*: i.e. a Shape**
  vector<Shape *>::const_iterator p;
  for (p=vs.begin(); p!=vs.end(); ++p) { // loop over the whole vector
    cout <<  (*p)->Tipo()
         << "   Area = "<< (*p)->Area() << endl ;
  }                     // area of each Shape

  cout << endl;
  return 0;
}
```

Vector of abstract class S

```
#include "Shape.h"
#include "Cerchio.h"
#include "Rettangolo.h"
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;

int main() {

  vector<Shape *> vs;            // a vect
  for (int i=0; i!=10; i++) {
    vs.push_back(new Cerchio(i));
    vs.push_back(new Rettangolo(i, i+1));
  }

  // iterator: essentially a pointer to Sh
  vector<Shape *>::const_iterator p;
  for (p=vs.begin(); p!=vs.end(); ++p) { /
    cout <<  (*p)->Tipo()
         << "   Area = "<< (*p)->Area() <<
  }                     // area of each Sha

  cout << endl;
  return 0;
}
```

```
nbacer(~/lez_9)>./vectshape
  Sto costruendo una Shape...
  Sto costruendo un Cerchio...
  Sto costruendo una Shape...
  Sto costruendo un Rettangolo...
  ...............
Cerchio   Area = 0
Rettangolo   Area = 0
Cerchio   Area = 3.14159
Rettangolo   Area = 2
Cerchio   Area = 12.5664
Rettangolo   Area = 6
Cerchio   Area = 28.2743
Rettangolo   Area = 12
Cerchio   Area = 50.2655
Rettangolo   Area = 20
Cerchio   Area = 78.5398
Rettangolo   Area = 30
Cerchio   Area = 113.097
Rettangolo   Area = 42
Cerchio   Area = 153.938
Rettangolo   Area = 56
Cerchio   Area = 201.062
Rettangolo   Area = 72
Cerchio   Area = 254.469
Rettangolo   Area = 90

nbseve(~/lez_9)>
```

vector: 4 - Le funzioni (algoritmi) di *vector*

Permettono di eseguire delle operazioni sul container.

http://www.cplusplus.com/reference/algorithm/

Reference **<algorithm>**

library

<algorithm>

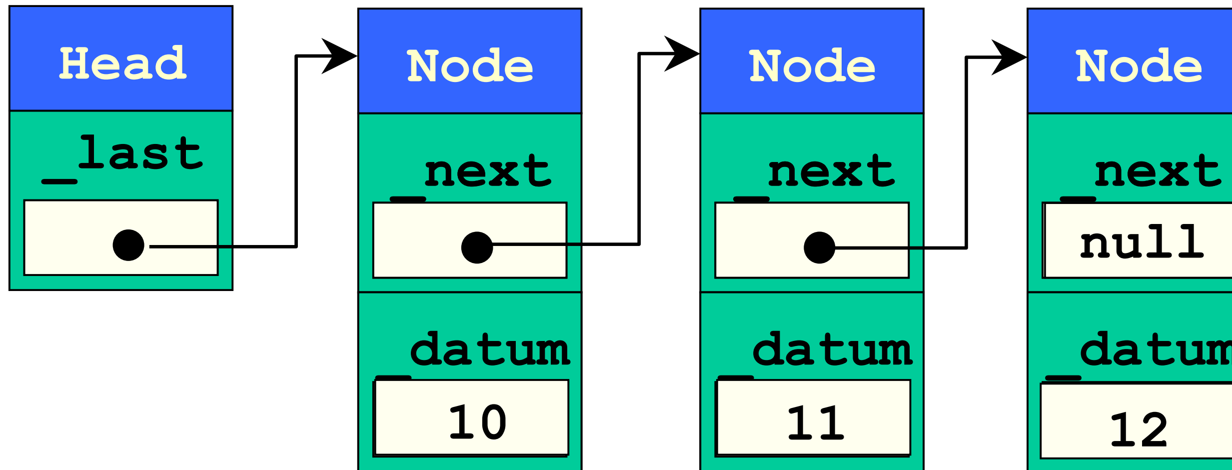
<algorithm>

Standard Template Library: Algorithms

The header `<algorithm>` defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers. Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

List



Si utilizza come vector (sostituendo a `vector` la parola `list`)

Piu' efficiente nel gestire le inserzioni di elementi

Map

Una "map" e' un container associativo costituito da coppie chiave-valore (key - T)

La map e' indicizzata rispetto alla chiave

Il valore della chiave deve essere univoco

Esempio: `map<"string", int>`

John	28

Mary	30

...

Stephen	28

Precisazione sull'uso di `const`

Per indicare che un metodo non puo' modificare gli attributi della classe, `const` deve essere posto **dopo** la definizione della classe (prima del ; nel .h o della { nel .cc)

```
bool operator > (const Shape &s2)  const ;
```

Shape.h

```
bool Shape::operator > (const Shape &s2)  const {


}
```

Shape.cc

Quando nei container STL si utilizza un const_iterator , non e' possibile modificare il valore a cui esso si riferisce

(un esempio nella prossima lezione)

22) L'algebra di una classe (cenni)

Overloading dei metodi (1)

Due metodi della stessa classe con lo stesso nome possono essere distinti in base

1. Al numero degli argomenti

```
CorpoCeleste() ;
CorpoCeleste(string nome, float mass) ;
CorpoCeleste(string nome, float mass, float x, float y);
```

2. Al tipo degli argomenti

```
calcolaPosizione(float fx,float dt);
calcolaPosizione(TwoVector f, float dt);
```

Overloading dei metodi (2)

3. [ovviamente:] Al tipo e al numero degli argomenti

```
calcolaPosizione(float fx, float fy, float dt);

calcolaPosizione(TwoVector f, float dt);
```

Non e' possibile distinguere tra metodi in base al **tipo** del metodo (void, int, double, ...)

Overloading degli Operatori

Inserire un'algebra all'interno di una classe

Un esempio: la classe TwoVector dei vettori in due dimensioni

- Operatori di assegnazione
- Operatori unari
- Operatori binari
- [Operatori ternari]

Overloading dell'Operatore << (cout)

Precisazioni sull'*Overloading* degli operatori (1)

1. Operatori membro (*member operators*)

Gli **operatori** definiti all'interno della classe si chiamano operatori membro

Gli operatori definiti in questo modo sono un modo naturale (e compatto) di identificare un metodo (della classe) e come tali possono accedere ai membri privati

Gli operatori membro si applicano ad oggetti di una classe (come i metodi) e possono avere come argomento

nessun oggetto (operatori unari) (es. ++ , - (segno))
un oggetto (operatori binari) (es. +, - (sottrazione))
[due oggetti (operatori ternari)]

…………………………………….

Precisazioni sull'*Overloading* degli operatori (2)

Esempi di operatori unari

```
a++
a.operator++()
```

Esempi di operatori binari

```
a+b
a.operator+(b)
```

```
a=b
a.operator=(b)
```

```
a=b+c
a.operator=(b.operator+(c))
```

Scritture Equivalenti!!!!

Precisazioni sull'*Overloading* degli operatori (3)

2. Operatori non membro (non *member operators*)

Gli operatori definiti all'esterno della classe si chiamano operatori non membro

Si usa per gli operatori che agiscono sugli oggetti della classe, ma sono applicati ad oggetti di un'altra classe.

```
cout << a

cout.operator<<(a)
```

<< e' un operatore che si applica all'oggetto cout

pero' qui fa riferimento alla classe a cui appartiene `a`, e lo implementero' alla fine dell'implementazione della classe a cui appartiene `a` (un esempio nel seguito)

Scritture Equivalenti!!!!

Un (primo esempio): la classe Shape (1)

Shape.h

```
................................
const bool diff_A(Shape &s2)    ;
................................
const bool operator != (Shape &s2) ;
................................
```

Shape.cc

```
................................
const bool Shape::diff_A(Shape* &s2) {
  return ( this->Area() != s2->Area() ) ;
}

const bool Shape::operator != (Shape &s2) {
  return ( this->Area() != s2.Area() ) ;
}
................................
```

Un (primo esempio): la classe Shape (2)

```
#include "Shape.h"
#include "Rettangolo.h"

#include <iostream>
using namespace std ;

int main() {

  Rettangolo alfa(2,3);
  Rettangolo beta(4,5);
  Rettangolo gamma = alfa ;

  cout << " " << (alfa!=beta)   <<
  cout << " " << (alfa!=gamma) <<

  cout << endl;

  return 1;

}
```

```
nbseve(~/Eserc_7)>./provop
  Sto costruendo una Shape...
  Sto costruendo un Rettangolo...
  Sto costruendo una Shape...
  Sto costruendo un Rettangolo...
 1
 0

  ... sto distruggendo un Rettangolo
  ... sto distruggendo una Shape
  ... sto distruggendo un Rettangolo
  ... sto distruggendo una Shape
  ... sto distruggendo un Rettangolo
  ... sto distruggendo una Shape
nbseve(~/Eserc_7)>
```

piu' naturale di

```
alfa.diff_A(beta)
```

TwoVector.h

```
// Questo file e' stato ottenuto modificando,
// per scopo didattico, il file originario
// ThreVector.h (e ThreeVector.icc)
// della libreria CLHEP del Cern
// ===========================================
// This file is a part of the CLHEP - a Class
// Library for High Energy Physics.
// .SS Authors Leif Lonnblad and Anders Nilsson.
// ============================================
//
// .SS Authors
// John Marraffino and Mark Fischler
//
```

```
// Attributi
 private:
   double dx, dy;    // The components.
```

```
public:

// Metodi
  …………………
```

```
// OPERATORI
```

```
// UNARI

   TwoVector operator - () const;

   // Unary minus.
```

```
// BINARI

   TwoVector & operator = (const  TwoVector &);
   // Assignment
```

```
TwoVector operator + (const TwoVector &);
  // Addition of 2-vectors.

TwoVector operator - (const TwoVector &);
  // Subtraction of 2-vectors.

TwoVector & operator += (const TwoVector &);
  // Addition ( += ).

TwoVector & operator -= (const TwoVector &);
  // Subtraction (-= ).

TwoVector & operator *= (float);
  // Scaling with real numbers.
```

```
// BOOLEANI ( Comparisons )
 bool operator == (const TwoVector &) const;

 bool operator != (const TwoVector &) const;
```

```
};
```

```
// dopo il segno   };  che indica la fine della
// dichiarazione della classe
```

```
TwoVector operator * (const TwoVector & , float );

TwoVector operator * (float , const TwoVector & );
```

```
// Overloading dell'operatore <<  (cout)
//
ostream & operator << (ostream &, const TwoVector &);
// Output to a stream.
```

```
#include “TwoVector.icc”

#endif  //TWOVECTOR_H
```

TwoVector.icc

```
// Questo file e' stato ottenuto modificando,
// per scopo didattico, il file // originario
// ThreeVector.icc (e ThreeVector.h) della
// libreria CLHEP del Cern
// =====================
// This file is a part of the CLHEP -
// a Class Library for High Energy Physics.
// This is the definitions of the inline
// member functions of the
// TwoVector class.
// =====================


 TwoVector TwoVector::operator - () const {
   return TwoVector(-dx, -dy);
 }

 TwoVector & TwoVector::operator = (const TwoVector & p) {
   dx = p.x();
   dy = p.y();
   return *this;
 }
```

```
TwoVector TwoVector::operator + (const TwoVector & a) {
   return TwoVector(dx + a.x(), dy + a.y());
}

TwoVector TwoVector::operator - (const TwoVector & a) {
   return TwoVector(dx - a.x(), dy - a.y());
}

TwoVector& TwoVector::operator +=  (const TwoVector & p) {
  dx += p.x();
  dy += p.y();
  return *this;
}

TwoVector& TwoVector::operator -= (const TwoVector & p) {
  dx -= p.x();
  dy -= p.y();
  return *this;
}
```

```
TwoVector& TwoVector::operator *= (float a) {
  dx *= a;
  dy *= a;
  return *this;
}

bool TwoVector::operator == (const TwoVector& v) const {
  if (v.x()==x() && v.y()==y()) {
    return true ;
  } else {
    return false ;
  }
}

bool TwoVector::operator != (const TwoVector& v) const {
  if (v.x()!=x() || v.y()!=y()) {
    return true ;
  } else {
    return false ;
  }
}
```

```
inline TwoVector operator * (const TwoVector & p, float a) {
  return TwoVector(a*p.x(), a*p.y());
}


inline TwoVector operator * (float a, const TwoVector & p) {
  return TwoVector(a*p.x(), a*p.y());
}
```

```
ostream & operator << (ostream & fstream, const TwoVector &
v) {

   fstream << " (" << v.x() << "," << v.y() << ") ";


}
```

non-member operators

Precisazione sull'uso di `const`

Per indicare che un metodo non puo' modificare gli attributi della classe, `const` deve essere posto **dopo** la definizione della classe (prima del ; nel .h o della { nel .cc)

```
bool operator > (const Shape &s2)   const ;
```

Shape.h

```
bool Shape::operator > (const Shape &s2)   const {


}
```

Shape.cc

Quando nei container STL si utilizza un const_iterator , non e' possibile modificare il valore a cui esso si riferisce

(un esempio nella prossima lezione)

23) L'algebra di una classe (precisazioni)

Precisazioni sull'*Overloading* degli operatori (1)

1. Operatori membro (*member operators*)

Gli **operatori** definiti all'interno della classe si chiamano operatori membro

Gli operatori definiti in questo modo sono un modo naturale (e compatto) di identificare un metodo (della classe) e come tali possono accedere ai membri privati

Gli operatori membro si applicano ad oggetti di una classe (come i metodi) e possono avere come argomento

nessun oggetto (operatori unari) (es. ++ , - (segno))

un oggetto (operatori binari) (es. +, - (sottrazione))

[due oggetti (operatori ternari) (es. ?)]

…………………………………….

Precisazioni sull'*Overloading* degli operatori (2)

Esempi di operatori unari

```
a++
a.operator++()
```

Esempi di operatori binari

```
a+b
a.operator+(b)
```

```
a=b
a.operator=(b)
```

```
a=b+c
a.operator=(b.operator+(c))
```

Scritture Equivalenti!!!!

Precisazioni sull'*Overloading* degli operatori (3)

Esempio di operatori ternari

```
a ? b: c
" a.operator?(b, c) "
```

Non implementabile in questa forma , ma questo sarebbe il significato

```
if(a) {
  return b;
 } else {
  return c;
}
```

L'operatore ternario non ammette overloading

Non puo' essere ridefinito

Esempi di operatori binari

```
a+b
a.operator+(b)
```

Precisazioni sull'*Overloading* degli operatori (4)

```
using namespace std;
#include <iostream>

int main(){
  int i ;
  int a = 2, b = 3;
  bool d;

  cout <<  endl <<      "a = " << a
                <<  ";   b = " << b  << endl << endl ;

  d = true;
  i = (d ? a : b);
  cout << "d = "<< boolalpha << d << "  i = " << i << endl << endl;

  d = false;
  i  = (d ? a : b);
  cout << "d = "<< boolalpha << d <<  " i = " << i << endl << endl;

  return 1;

}
```

```
nbtmx5(~)>g++ qm.cpp -o qm
nbtmx5(~)>./qm
a = 2;   b = 3
d = true  i = 2
d = false i = 3
```

Precisazioni sull'*Overloading* degli operatori (5)

```
using namespace std;
#include <iostream>

int main(){
  int i ;
  int a = 2, b = 3;
  bool d;

  cout <<  endl <<       "a = " << a
                <<  ";   b = " << b  << endl << endl ;

  d = true;
           //i = (d ? a : b); Equivalente a
  if(d) {
    i = a;
   } else {
    i = b;
  }



 ....continua....
```

Precisazioni sull'*Overloading* degli operatori (6)

```
nbtmx5(~)>g++ qm2.cpp -o qm2
nbtmx5(~)>./qm2
a = 2;   b = 3
d = true  i = 2
d = false i = 3
```

```
......continua....

  d = false;
            //i = (d ? a : b); Equivalente a
  if(d) {
    i = a;
   } else {
    i = b;
  }

  cout << "d = "<< boolalpha << d << "  i = " << i << endl << endl;


  return 1;

}
```

Precisazioni sull'*Overloading* degli operatori (6)

Non e' possibile l'overloading dell'operatore ?

Per effettuare l'overloading e' necessario che:

- o si tratti di un member operator
- oppure che il non member operator abbia come argomento almeno un oggetto definito dall'utente

Esiste una lista degli operatori (di cui si puo' effettuare l'overload):

http://en.cppreference.com/w/cpp/language/operators

http://stackoverflow.com/questions/4421706/operator-overloading/4421708#4421708

operator overloading https://en.cppreference.com/w/cpp/language/operators/

<table>
<tr><th colspan="7">Common operators</th></tr>
<tr><th>assignment</th><th>increment
decrement</th><th>arithmetic</th><th>logical</th><th>comparison</th><th>member
access</th><th>other</th></tr>
<tr>
<td>a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b</td>
<td>++a
--a
a++
a--</td>
<td>+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b</td>
<td>!a
a && b
a || b</td>
<td>a == b
a != b
a < b
a > b
a <= b
a >= b
a <=> b</td>
<td>a[b]
*a
&a
a->b
a.b
a->*b
a.*b</td>
<td>a(...)
a, b
? :</td>
</tr>
<tr><th colspan="7">Special operators</th></tr>
<tr><td colspan="7">static_cast converts one type to another related type
dynamic_cast converts within inheritance hierarchies
const_cast adds or removes cv qualifiers
reinterpret_cast converts type to unrelated type
C-style cast converts one type to another by a mix of static_cast, const_cast, and reinterpret_cast
new creates objects with dynamic storage duration
delete destructs objects previously created by the new expression and releases obtained memory area
sizeof queries the size of a type
sizeof... queries the size of a parameter pack (since C++11)
typeid queries the type information of a type
noexcept checks if an expression can throw an exception (since C++11)
alignof queries alignment requirements of a type (since C++11)</td></tr>
</table>

References

1. ↑ Operator Overloading (http://stackoverflow.com/questions/4421706/operator-overloading/4421708#4421708) on StackOverflow C++ F

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/language/operators&oldid=110038"

Precisazioni sull'*Overloading* degli operatori (8)

```
using namespace std;
#include <iostream>

// definisco una classe
class Complex {

  protected:
    double _ra;
  public:
    Complex() {};
    Complex(double a) { _ra=a;};
    double Ra() const {return _ra;};

    operator int() {return (int) this->Ra() ;}
};

// definisco un operator non member che implementa la somma
int operator+(int const & a, Complex const & b)    {

        return a + b.Ra();
}
```

Precisazioni sull'*Overloading* degli operatori (9)

```
//int operator?(bool const &d, int const & a, Complex const & b)    {
//
//                return a + b.Ra();
//}  // NON E' POSSIBILE L'OVERLOADING DELL'OPERATORE ?

// E' POSSIBILE DEFINIRE UNA FUNZIONE CON PIU' ARGOMENTI
int pippo(double const & a, double const &b, double const &c, double
const & d) {

         return a + b + c + d;

}

//inizia il programma main

int main(){
           int i , a = 2;
           Complex b(3);
```

Precisazioni sull'*Overloading* degli operatori (10)

```
nbtmx5(~)>g++ qm3.cpp -o qm3
nbtmx5(~)>./qm3
a = 2;   b = 3

8
```

```
              cout <<  endl <<      "a = " << a
//                 <<  ";   b = " << b.Ra()  << endl << endl ;
                   <<  ";   b = " <<      b  << endl << endl ;

              // i = a + b;
              i = operator+(a,b);


              cout << endl << pippo(1,2,2.5,3.4) << endl ;

              return 1;
}
```