I numeri complessi

Stefano Lacaprara

Cosa vogliamo fare

- C++ possiede (pochi) tipi elementari per
  - Int
  - Float
  - Double
  - Char
  - Bool

- Ma se dobbiamo lavorare con I numeri complessi?
  - (in realta' c'e' qualcuno che ha gia' implementato i numeri complessi in C++, basta fare #include <complex> ma noi facciamo finta di niente...)

Soluzione elementare

- Per ogni numero complesso a, definisco
  - Double aReal,aImm;
- E faccio tutto o a mano
- Se b = a*a
- bReal=aReal*aReal-aImm*aImm ;
- Ecc ecc

- Così si fa in fortran (o in c)
- Così abbiamo fatto tutti fino ai linguggi ad oggetti

Scopo dell’ esercizio

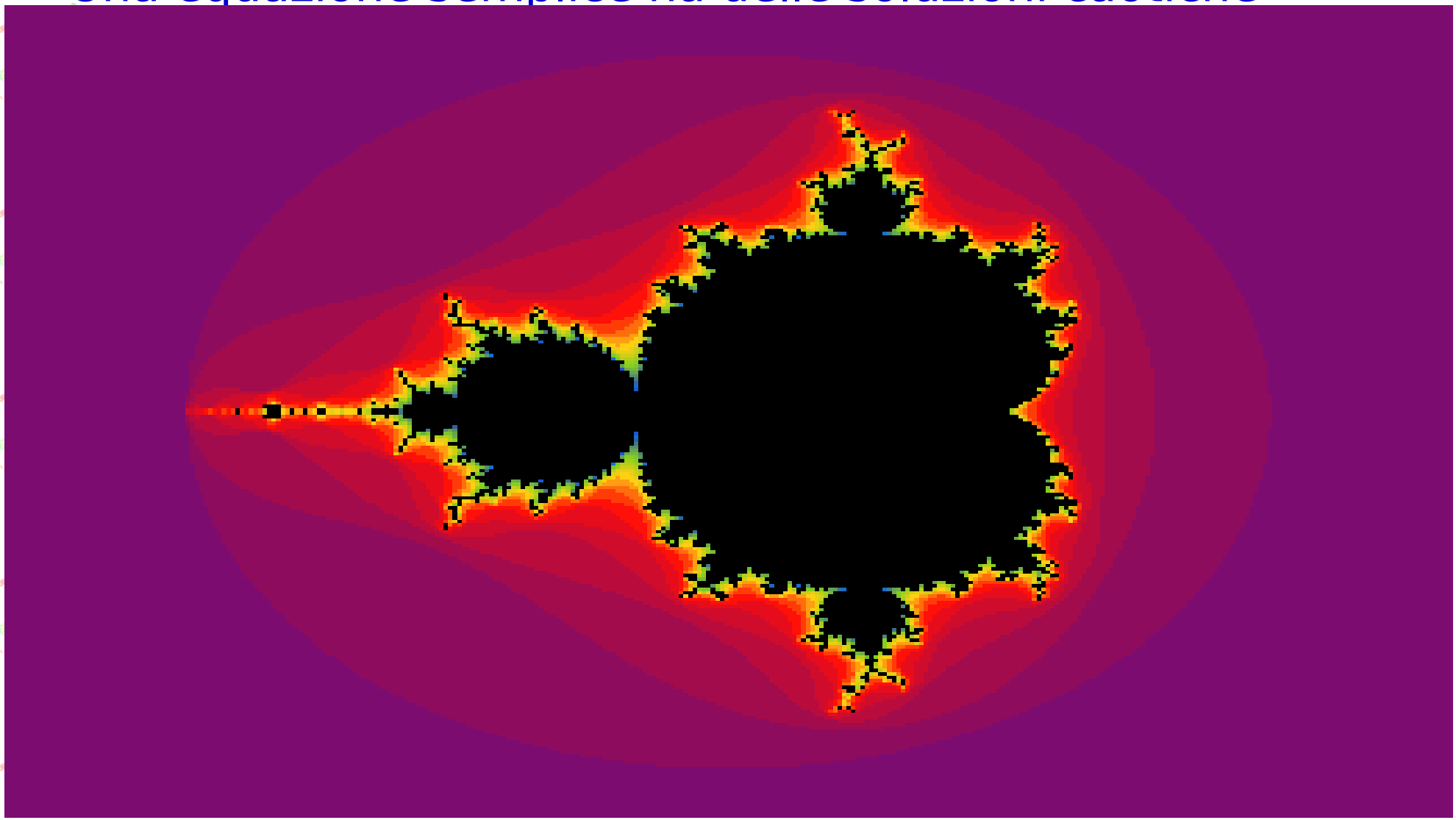
- Definire un nuovo tipo di variabili complesse
- Definire le operazioni elementari
  - Complex a,b;
  - Definire una somma per esempio, un prodotto ecc ecc
    - Lo posso fare con delle funzioni (C like)
      - Complex somma(Complex a, Complex b);
        - Complex c=somma(a,b);
    - Lo posso fare con dei metodi della classe Complex
      - Cioè, un numero complesso sa come sommarsi ad un altro
      - Complex Complex::somma(Complex a);
        - Complex c=b.somma(a);
    - Posso estendere il C++
      - Complex c=a+b;

Diamo un senso al tutto

- Insieme di Mandelbrot
  - Lo definiamo
  - Lo guardiamo
- Problema:
  - Quanto vale la sua area
  - Problema matematico (credo) non risolto...
  - Bene, lo calcoliamo noi con il metodo MonteCarlo!

Mandelbrot ...

- Una delle scoperte più affascinanti della matematica negli ultimi 2 secoli
- Una equazione semplice ha delle soluzioni caotiche



definizione

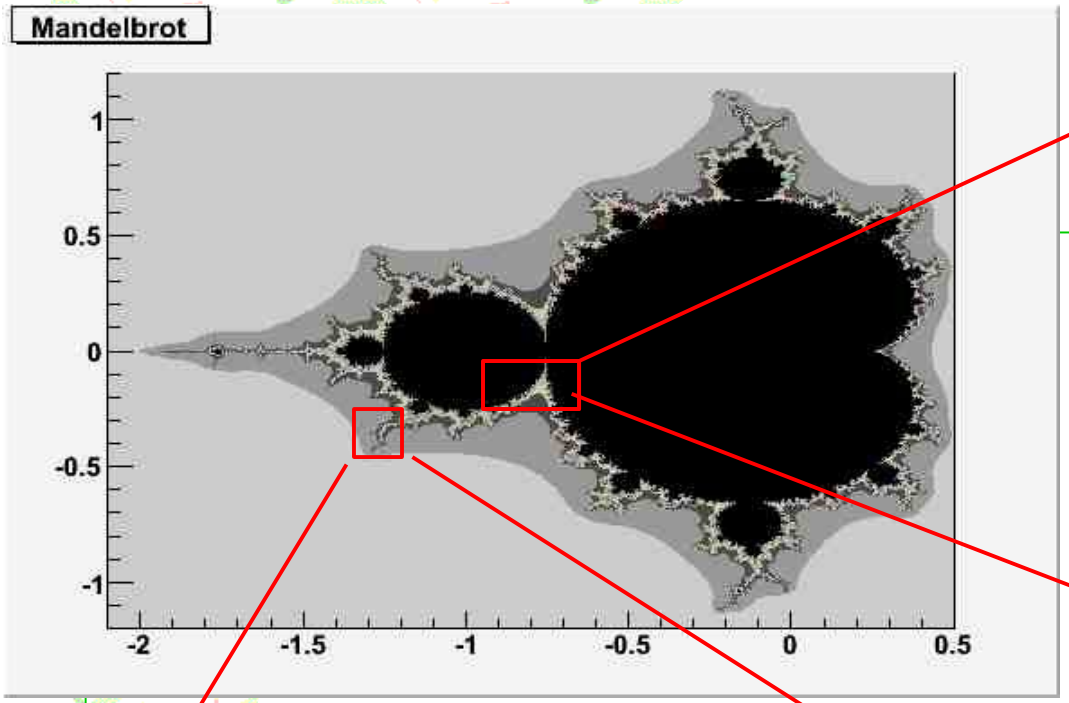
- Consideriamo il piano complesso, in cui un punto (x,y) può essere rappresentato con un numero
  - c=x+iy
- e consideriamo la seguente iterazione
  - z=0+i0

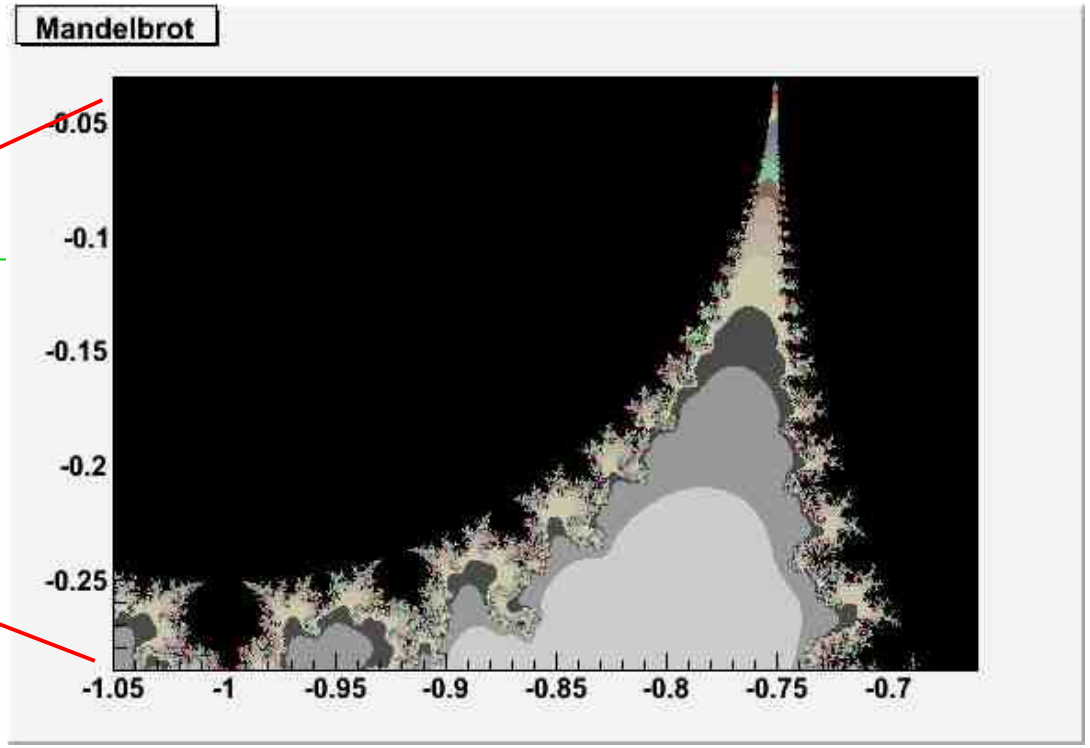
z(i+1)=z(i)*z(i)+c

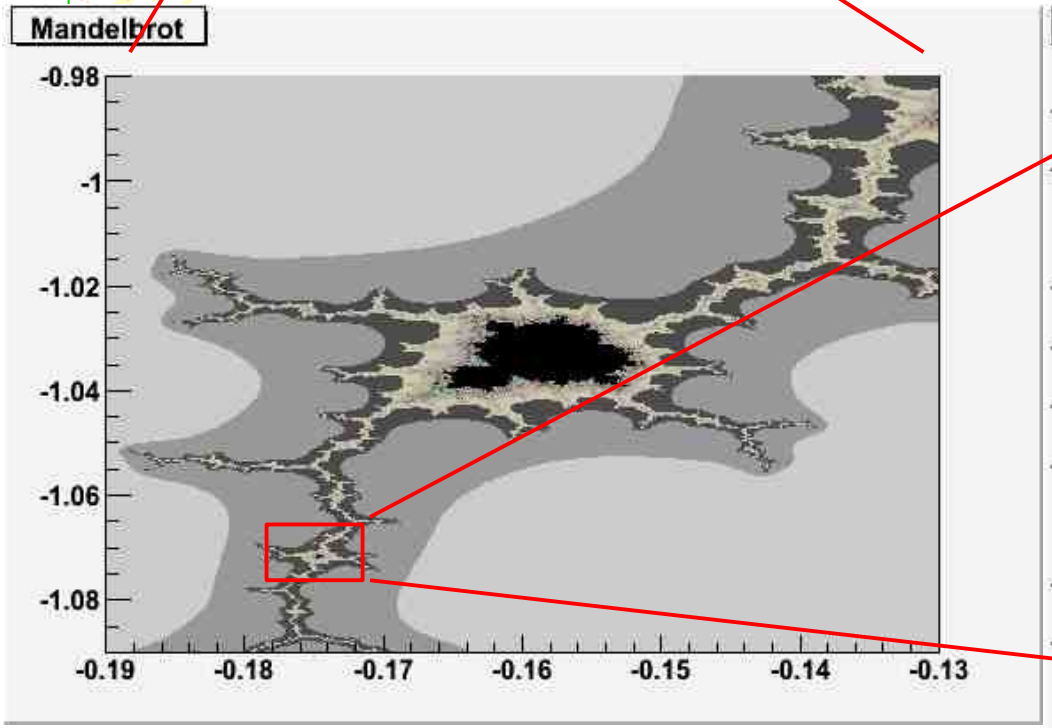
- Cosa succede a z? (NB z e' un complesso!)
  - il modulo di z diverge
  - il modulo di z non diverge

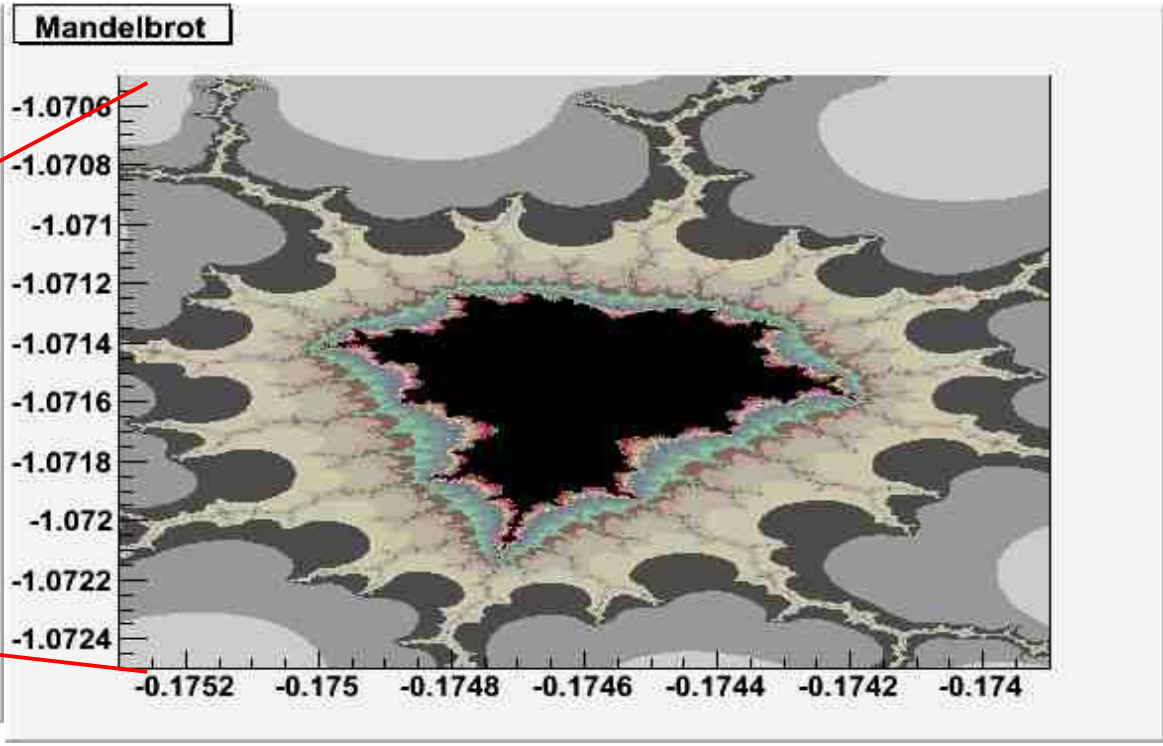
Insieme di Mandelbrot

- Definizione:
  - l'insieme di mandelbrot è costituito da quei punti per cui z non diverge (la parte in nero del disegno di prima)
- Per I punti in cui diverge, posso mettere un colore che rifletta dopo quanti cicli diverge
  - Matematicamente divergere e' un concetto limite (z->∞), dal punto di vista computazionale decido che z diverge se |z| diventa piu' grande di una qualche soglia
  - "z Non diverge" se dopo un certo numero di cicli e' ancora minore della soglia
- Come per molte altre cose, la frontiera e' la parte piu' interessante...


Mandelbrot
1
0.5
0
-0.5
-1
-2
-1.5
-1
-0.5
0
0.5


Mandelbrot
-0.05
-0.1
-0.15
-0.2
-0.25
-1.05
-1
-0.95
-0.9
-0.85
-0.8
-0.75
-0.7


Mandelbrot
-0.98
-1
-1.02
-1.04
-1.06
-1.08
-0.19
-0.18
-0.17
-0.16
-0.15
-0.14
-0.13

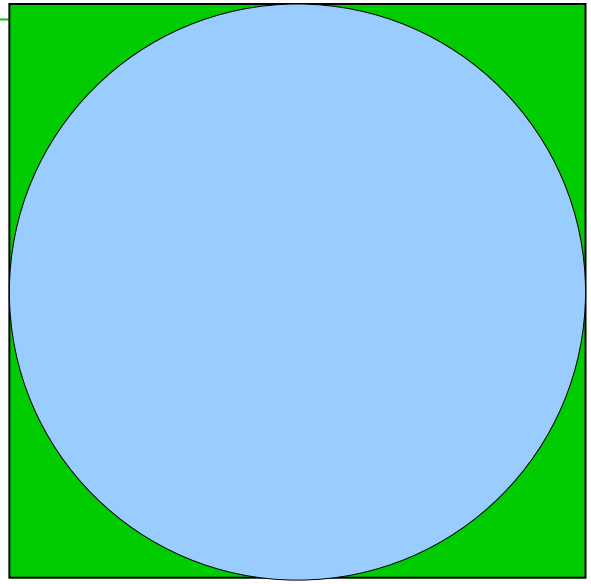

Mandelbrot
-1.0706
-1.0708
-1.071
-1.0712
-1.0714
-1.0716
-1.0718
-1.072
-1.0722
-1.0724
-0.1752
-0.175
-0.1748
-0.1746
-0.1744
-0.1742
-0.174

Visualizzarlo?

- Lo facciamo con ROOT, appena ve lo ho spiegato (tra un po'...)
- Per il momento affrontiamo un problema più semplice:
  - quale è l'area dell'insieme di mandelbrot (dei punti in cui non diverge...)???
- Problema insoluto matematicamente
  - "vicino a 1.54"
- Calcoliamolo!

Metodo MonteCarlo (o del teppista)

- Approccio probabilistico
- voglio calcolare l'area di un cerchio
- lancio a caso dei sassi sul tavolo, e conto quante volte becco il cerchio
- Alla fine (per un numero abbastanza alto di lanci... vedremo in seguito...)



$$\frac{AreaCerchio}{AreaQuadrato} = \lim_{n \to \infty} \left(\frac{sassiNelCerchio}{SassiTotali} \right) = \left(\frac{PiGreco}{4} \right)$$

- Posso calcolare pi greco!

Numero Complesso: la classe

```
class  Complex{
public:
  Complex(double r,double i)     {realpart=r;
immpart=i;}
  double real(){return realpart;}
  double imag(){return immpart;}
private:
  double immpart,realpart;
};
```

definisco la classe

costruttore

metodi per avere parti reale e immaginaria

dati privati per contenere l'informazione

Un po’di cose …

- Io definisco
  - **double real(){return realpart;}**
  - ok perché è un metodo veloce, in realtà quello che si usa fare è mettere in un posto (= in un file) le definizioni dei metodi e della classe, in un altro il codice (implementazione)
- Scriverò da una parte la dichiarazione
  - **double real() ;**
  - e dall’altra l’implementazione
  - **double Complex::real(){return realpart;}**

Esempio

```
class  Complex{
public:
  Complex(double,double);
  double real();
  double imag();
private:
  double immpart,realpart;
};
Complex::Complex(double r, double i) {realpart=r; immpart=i;}
double Complex::real(){return realpart;}
double Complex::imag(){return immpart;}
```

Uso Complex in main

- Poi ci manca il main

gli diciamo di usare le classi definite in Complex.h

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

using namespace std;

int main(){
  Complex c1(1,1);
  Complex c2(2,-3);
  Complex c3(4.);
  cout<<“Ecco “<<
     c1.real()<<“ “ <<
     c1.imag()<<endl;

}
```

- Infine devo compilare

Complex.h e Complex.cc

- Per una class facciamo 2 files (che hanno lo stesso nome della classe, per semplicita')
  - **Complex.h**
    - Definizione della class e dichiarazione metodi: niente codice a parte quello molto semplice.
  - **Complex.cc**
    - Contiene l'implementazione dei metodi (#include “Complex.h”)
- Poi abbiamo un **main.cpp** (cpp cosi' vedo subito che non contiene una classe) che usa Complex (#include “Complex.h”)
- Compilare:
  - g++ -c Complex.cc (creo Complex.o compilazione)
  - g++ -o main main.cpp Complex.o (creo eseguibile – link)

Proteggere da inclusioni multiple

- Che succede se includo due volte lo stesso Complex.h (anche in modo indiretto)?
- E' come se definissi piu' volte la stessa classe, non si puo'!
- Mi proteggo: in OGNI file.h metto, rigorosamente

```
#ifndef CLASSNAME_H
#define CLASSNAME_H

/// Qui ci va il codice
class ClassName {
};

#endif
```

un po’ di cose che ci serviranno

- Parametri di default
- Il solo costruttore che ho definito vuole due parametri
- Potrei definirne uno senza parametri, che definisce automaticamente 0+0i
- Oppure uno con un parametro per definire oggetti del tipo 5+0i

- Lo posso fare in modo compatto

```
Complex(double r=0, double i=0);

...

Complex::Complex(double r, double i) {realpart=r;
immpart=i;}
```

- Cosi' ho anche un costruttore di default

const … per evitare gli errori

```
class  Complex{
public:
  Complex(double,double);
  double real() const ;
  double imag() const ;
private:
  double immpart,realpart;
};
Complex::Complex(double r, double i) {realpart=r;
immpart=i;}
double Complex::real() const {return realpart;}
double Complex::imag() const {return immpart;}
```

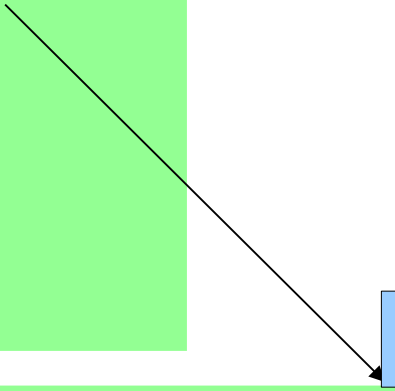
così facendo informiamo il compilatore che questi metodi non potranno cambiare l'oggetto, cioè non potranno cambiare parte reale e immaginaria

Adesso i metodi ...

- Dato un numero immaginario, gli dobbiamo dare la capacità di sommarsi ad un altro

Complex.h

```
double mod();
double mod2();
Complex somma(Complex in) const ;
Complex sottrai(Complex in) const ;
Complex moltiplica(Complex in) const ;
Complex dividi(Complex in) const ;
Complex inverso() const ;
```



Complex.cc

```
Complex Complex::somma(Complex in) const {
  return Complex (realart+in.real(),
                  immpart+in.imm());
}
```

quindi...

- Proviamo ad usarle

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

using namespace std;

int main(){
  Complex c1(1,1);
  Complex c2(2,-3);
  cout<<“Ecco “<<
     c1.somma(c2).real()<<“ “ <<
     c1.somma(c2).imag()<<endl;
}
```

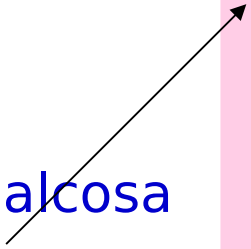
perfetto, ma ...

- non è proprio quello che volevamo raggiungere
  - abbiamo adesso il modo di sommare, sottrarre due numeri ecc
  - ma non abbiamo davvero esteso il C++:
  - se a e b sono interi, posso fare “a+b”
  - ora se ca e cb sono complessi, non posso fare “ca+cb”
- Come si fa?
  - quello che io vorrei alla fine è qualcosa del tipo

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

using namespace std;

int main(){
  Complex c1(1,1);
  Complex c2(2,-3);
  Complex c3 = c1+2/c2;
  cout<<“Ecco “<<c3<<endl;
}
```



quindi ...

- Mi serve:
  - un modo per dire al compilatore come sommare sottrarre ecc numeri complessi fra loro
  - come fare lo stesso fra complessi e interi e float
  - come stampare un numero complesso

  - ... l'overloading degli operatori +- ... e del comportamento sotto cout!
  - Overloadig:
    - Ci sono due (o piu') funzioni con lo stesso nome ma che hanno argomenti di diverso tipo somma(int, int), somma(complex, int)
    - Il C++ prende la funzione giusta a seconda degli argomenti che gli passiamo (non si puo' in C)

C++

- Posso definire delle funzioni chiamate
- operator*(...)
- operator+(...)
- ecc che estendono io significato di *,+ ecc
- si chiamano operatori simmetrici, perché agiscono su due elementi

```
Complex operator+(Complex a, Complex b){
    return Complex(a.real()+b.real(), a.imag()+b.imag()));
}
```

Da notare…

- Questo operatore NON appartiene alla classe Complex, è una funzione (esterna) C like
- Ho definito la somma di due complessi, d'ora in poi il C++ se gli chiedo
  - a+b in realtà farà
  - operator+(a,b) automaticamente

Un problema ...

- ma se voglio sommare un intero a un complesso?

- Facile soluzione
  - Complex operator+(int a, Complex b)...
- Questo permette di fare 4+a, ma in teoria non a+4
  - il c++ non si permete di assumere che la somma debba sempre essere commutativa....

- Dovei quindi definire a+X e X+a per ogni a
  - int
  - float
  - double
  - ...

con una soluzione ...

- noi sappiamo che è sempre possibile passare da intero a float, da float a double ecc ecc
  - il compilatore lo fa da solo!
- posso fare lo stesso in C++
  - spiegare al compilatore che una variabile double può sempre essere automaticamente convertita in un Complex
    - poi tutto il resto è automatico, perchè se ho un int lui sa già come passare a double
- Lo abbiamo già fatto!!

Costruttore

Complex(double r=0, double i=0);

- quindi, se ho un double, lo posso convertire automaticamente in Complex usando i parateri di default
- Dovrei fare
  - Complex(7)+a
- ma in realtà non serve perché il compilatore capisce che l'unica cosa che sa sommare ad un complesso è un complesso, e si accorge che sa convertire un double in un complesso anche se non glielo dico esplicitamente!

Quindi ...

- Se definisco +*-/ decentemente, posso fare

```
int main(){
  Complex c1(1,1);
  Complex c2(2,-3);
  Complex c3(4.);
  cout <<" Modulo di c1 "<<c1.mag()<<endl;
  cout <<" Modulo di c3 "<<c2.mag()<<endl;
  cout <<" Coniugato di c1 "<<c1.coniugate().real()
       <<" "<<c1.coniugate().imag()<<endl;
  cout<< "C1 "<<c1<<endl;
  cout<< "C2 "<<c2<<endl;
  cout <<" c1 piu' c2 "<<(c1+c2).real()<<" "<<(c1+c2).imag()<<endl;
  cout <<" c1 meno c2 "<<(c1-c2).real()<<" "<<(c1-c2).imag()<<endl;
  cout <<" c1 * c2 "<<(c1*c2).real()<<" "<<(c1*c2).imag()<<endl;
  cout <<" c2 * 5 "<<(c2*5)<<" "<<(c2+5)<<endl;
  cout <<" 5 * c2 "<<(c2*5)<<" "<<(5.+c2)<<endl;
  cout <<" 1/c2 "<<c2.inverse()<<" " <<1/c2<<" "
<<c2*c2.inverse()<<endl;
}
```

Come lo stampo?

- Ridefinisco l'operatore << !
- Vado a vedere come è definito normalmente e ne definisco uno simile per il tipo Complex

- std::ostream& operator<<(std::ostream&, Complex);

```
std::ostream& operator<<(std::ostream& os, Complex  c){
  os << '(' << c.real() << '+i' << c.imag() << ')';
  return os;
}
```

Numeri (pseudo)casuali

- Per usare il metodo MonteCarlo, abbiamo bisogno di un generatore di numeri casuali.
- Il C++ ne ha uno (di qualita' abbasanza scadente, ma per noi puo' andare bene)
  - rand()
  - Restituisce un numero intero pseudo casuale tra 0 e RAND_MAX (numeraccio molto grande)
  - Per avere un numero tra 0 e 1
    - float r = rand()*1./ RAND_MAX; // NB 1.!!
- Non usatelo per cose serie (fa veramente schifo...)
- Ne vedremo uno migliore con Root piu' avanti.

Prego...