

Sommario

- Parola chiave `friend`:
 - Funzioni e metodi .
- Operatori sovraccaricati:
 - `operator-()`
 - `operator+()`
 - `operator==()`
 - `operator<<()`
 - `operator=()`
 - `operator[]()` e `operator()()`
- Metodi indispensabili
 - Costruttori, distruttore, costruttore per copia e operatore di assegnamento.
- Esempio di tipo definito dall'utente, la classe `FloatArray`

Funzioni friend

- Il meccanismo delle funzioni `friend` consente ad una classe di accordare ad alcune funzioni l'accesso ai suoi membri non pubblici.
- La dichiarazione di una funzione `friend` comincia con la parola chiave `friend` e può comparire solo in una definizione di classe.
- Si possono anche dichiarare `friend` funzioni membro di un'altra classe definita precedentemente o un'intera classe (in tal caso tutti i metodi hanno l'accesso ai membri non pubblici della classe che accorda la qualifica di `friend`).

Funzioni friend

- In generale, si deve cercare di minimizzare il numero di funzioni che hanno accesso alla rappresentazione interna di una classe.
- L'uso delle dichiarazioni `friend` è comune con gli operatori sovraccaricati.
- Si utilizzano anche se si devono manipolare oggetti di classi distinte o per operazioni di uso generale.
- Vediamo un esempio in cui si implementa una funzione per copiare un oggetto di tipo `Pixel` in un oggetto di tipo `Point`.

Funzioni friend: esempio

Point.h

```
#ifndef POINT_H
#define POINT_H
class Pixel;
class Point{
    double x,y;
public:
    Point(double a, double b);
    ~Point();
    void Print()const;

    friend void Pixel2Point(
Point& po, const Pixel& pi);
};
#endif
```

Point.cpp

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

Point::Point(double a,double b)
: x(a),y(b){
    cout<<"Point()"<<endl;
}
Point::~~Point(){
    cout<<"~Point()"<<endl;
}
void Point::Print() const{
    cout<<"Point: ("<<x<<" ,
"<<y<<)"<<endl;
}
```

Per semplicità non sono stati implementati i metodi `Set` e `Get` per leggere e scrivere lo stato degli oggetti.

Funzioni friend: esempio

Pixel.h

```
#ifndef PIXEL_H
#define PIXEL_H
class Point;
class Pixel{
    int r,c;
public:
    Pixel(int a, int b);
    ~Pixel();
    void Print() const;

    friend void Pixel2Point(
        Point& po, const Pixel& pi);
};
#endif
```

Pixel.cpp

```
#include "Pixel.h"
#include <iostream>
using namespace std;
Pixel::Pixel(int a, int b):
    r(a),c(b){
    cout<<"Pixel()"<<endl;
}
Pixel::~Pixel(){
    cout<<"~Pixel()"<<endl;
}
void Pixel::Print() const{
    cout<<"Pixel: ("<<r<<"",
    "<<c<<"")<<endl;
}
```

Per semplicità non sono stati implementati i metodi Set e Get per leggere e scrivere lo stato degli oggetti.

Funzioni friend: esempio

Funzioni.cpp

```
#include "Point.h"
#include "Pixel.h"
void Pixel2Point(Point& po, const Pixel& pi){
    po.x = pi.c;
    po.y = pi.r;
}
```

Test.cpp

```
#include "Point.h"
#include "Pixel.h"
#include <iostream>
using namespace std;
int main(){
    Pixel pil(1,2);
    Point pol(-3.1,0);
    pil.Print();
    pol.Print();

    Pixel2Point(pol,pil);
    pol.Print();
    return 0;}

```

La funzione può accedere ai membri privati, in tal modo si risparmiano quattro chiamate a metodi: due Get sull'oggetto Pixel e due Set sull'oggetto Point

Una possibile uscita

```
Pixel()
Point()
Pixel: (1, 2)
Point: (-3.1, 0)
Point: (2, 1)
~Point()
~Pixel()
```

Operatori sovraccaricati

- L'*overloading degli operatori* permette al programmatore di definire versioni specifiche degli operatori predefiniti che usino come operandi oggetti di una classe.
- In tal modo si rende la *manipolazione degli oggetti* altrettanto *intuitiva* di quella dei tipi predefiniti.
- Si pensi, per esempio, alla possibilità di sottrarre due oggetti utilizzando l'operatore -, oppure di visualizzare un oggetto attraverso l'operatore <<.

```
Immagine a("im1.pgm"),b("im2.pgm"),c;
c=a-b;
cout<<c;
```

Operatori sovraccaricati

- Un operatore sovraccaricato è dichiarato nel corpo della classe allo stesso modo di una normale funzione membro, il suo nome è costituito dalla parola chiave operator seguito da uno degli operatori del C++.

```
const Immagine operator-(const Immagine &o) const;
```

- Alcune regole:
 - È possibile sovraccaricare solo operatori predefiniti.

Operatori sovraccaricati

- Il significato di un operatore non può essere cambiato per i tipi predefiniti. Si può solo sovraccaricare operatori per operandi di una classe.
- La precedenza predefinita non può essere modificata.
- La *arità* (numero di argomenti) predefinita degli operatori va conservata.
- Non possono essere sovraccaricati i seguenti operatori: `::` `.*` `.` `?:`

Operatori sovraccaricati

- Per definire *quali operatori offrire* è buona norma: (1) definire l'*interfaccia pubblica* della classe (*quali operazioni deve offrire la classe ai suoi utenti*); (2) scegliere quali metodi possono essere più semplici ed intuitivi da usare se definiti come operatori.
- Ogni operatore ha associato un *significato* che gli deriva dal suo uso con i tipi predefiniti: è importante che *non sia ambiguo* il suo significato sovraccaricato.
- Per esempio un metodo per la verifica dello stato `isEmpty()` potrebbe essere sostituito da `operator!()`, oppure `isEqual()` con `operator==()` e `copy()` con `operator=()`.

Operatori sovraccaricati

- Un *operatore* sovraccaricato è considerato *membro* di una classe solo quando viene usato con un *operando sinistro* oggetto della classe.
- Se l'operando sinistro non è oggetto della classe (si pensi all'operatore `<<`: per esempio `cout<<c;`) allora si deve dichiarare l'operatore *friend*.
- Devono essere membri i seguenti operatori:
`=` `[]` `()` `->`

Operatori sovraccaricati: esempio

- Introduciamo alcuni operatori che agiscono per una semplice classe `Complex` che rappresenta un numero complesso in virgola mobile: in tal modo possiamo porre attenzione alle dichiarazioni degli operatori, mentre il loro significato è noto.
- Descriviamo ogni operatore: dal comportamento con i tipi predefiniti definiamo quello per i tipi definiti dall'utente.

Operatori sovraccaricati: -

- Operatori *unari*:

```
int a=3;
cout<< -a;
```

- `operator-()`, non modifica l'oggetto quindi è `const` e ritorna un nuovo oggetto della stessa classe con il segno cambiato. L'oggetto ritornato può essere `const` per evitare che vi si invii un messaggio la cui azione potrebbe andare persa.

Operatori sovraccaricati: +

- Operatori *binari*:

TIPO PREDEFINITO

```
int a=3,b=5,c;
c=a+b;
```

- `operator+()`, crea un nuovo valore somma, quindi ritorna un nuovo oggetto della stessa classe. Non modifica gli oggetti operandi quindi possono essere `const` (e *reference*).

TIPO DEFINITO DALL'UTENTE

```
Immagine a("im1.pgm"),b("im2.pgm"),c;
c=a+b;
c=a.operator+(b);
```

Operatori sovraccaricati: ==

TIPO PREDEFINITO

```
int a=1,b=4;
if(a==b) ...
```

- `operator==()`, verifica se due oggetti sono uguali, quindi torna un `bool`. Non modifica gli oggetti operandi quindi possono essere `const` (e *reference*).

TIPO DEFINITO DALL'UTENTE

```
Immagine a("im1.pgm"),b("im2.pgm");
if(a==b) ...
if(a.operator==(b)) ...
```

Operatori sovraccaricati: <<

TIPO PREDEFINITO

```
int a=3;
cout<< a;
```

- `operator<<()`: deve agire su *reference* dello *stream* di uscita per poterlo modificare, inoltre deve tornare un *reference* allo stesso *stream* per permettere la concatenazione delle espressioni di stampa. Non modifica l'oggetto che si vuole stampare, quindi può essere `const` (e *reference*).

TIPO DEFINITO DALL'UTENTE

```
Immagine a("im1.pgm");
cout<<a;
operator<<(cout,a);
```

Operatori sovraccaricati: Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H
#include <iostream>

class Complex{
    double re,im;

public:
    Complex(double r=0, double i=0);
    ~Complex();
    void SetReIm(double r, double i);
    void GetReIm(double *r, double *i) const;

    const Complex operator-() const;
    const Complex operator+(const Complex &o) const;
    bool operator==(const Complex &o) const;

    friend std::ostream& operator<<(std::ostream& os, const Complex &o);
};
#endif
```

Operatori sovraccaricati: Complex.cpp

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

Complex::Complex(double r, double i){
    re=r; im=i;
    cout<<"Complex()"<<endl;
}
Complex::~Complex(){
    cout<<"~Complex()"<<endl;
}

void Complex::SetReIm(double r, double i){
    re=r; im=i;
}
void Complex::GetReIm(double *r, double *i) const{
    *r=re; *i=im;
}
```

Il metodo SetReIm modifica lo stato cambiando il valore della parte reale e immaginaria

Il metodo GetReIm legge lo stato, assegnando il valore della parte reale e immaginaria ai parametri puntatore

Operatori sovraccaricati: Complex.cpp

```
const Complex Complex::operator-() const{
    Complex tmp;
    tmp.re=-re;
    tmp.im=-im;
    return tmp;
}

const Complex Complex::operator+(const Complex &o) const{
    Complex tmp;
    tmp.re= re + o.re;
    tmp.im= im + o.im;
    return tmp;
}
```

Il metodo operator- crea un oggetto temporaneo, ne modifica lo stato e lo ritorna. Viene invocato il costruttore per copia durante il ritorno.

Il metodo operator+ crea un oggetto temporaneo, ne modifica lo stato e lo ritorna. I nuovi valori sono la somma dello stato dell'oggetto corrente (operando di sinistra) e lo stato dell'oggetto o (operando di destra).

Operatori sovraccaricati: Complex.cpp

```
bool Complex::operator==(const Complex &o) const{
    bool tmp;
    tmp= (re==o.re) && (im==o.im);
    return tmp;
}

ostream& operator<<(ostream& os, const Complex &o){
    os<<o.re<<" " <<showpos<<o.im<<"i";
    return os;
}
```

Il metodo operator== verifica lo stato dei due oggetti (operando di sinistra e di destra, o)

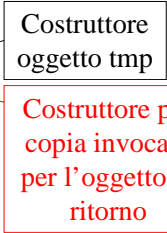
La funzione operator<< (non è membro della classe) stampa lo stato dell'oggetto (operando di destra, o) e ritorna lo stream di uscita. showpos è un manipolatore per modificare lo stato di formattazione dell'uscita: visualizza il segno + nella stampa dei valori positivi.

Operatori sovraccaricati: Test.cpp

```
void test(){
    Complex c1, c2(1,2);
    c1.SetReIm(-3,3);
    double r,i;
    c2.GetReIm(&r,&i);
    cout<<"c2: " <<r<<" " <<i<<endl;
    cout<<"-----" <<endl;
    cout<< -c2 <<endl;
    cout<<"-----" <<endl;
    c1=c1+c2;
    cout << c1 <<endl;
    cout<<"-----" <<endl;
    if (c1==c2)
        cout<<"uguali" <<endl;
    else
        cout<<"diversi" <<endl;
    cout<<"-----" <<endl;
}
```

Una possibile uscita

```
Complex()
Complex()
c2: 1 2
-----
Complex()
~Complex()
-1 -2i
~Complex()
-----
Complex()
~Complex()
~Complex()
-2 +5i
-----
diversi
-----
~Complex()
~Complex()
```



Operatori sovraccaricati: =

- L'assegnamento di un oggetto ad un altro della sua classe è gestito dall'*assegnamento di default membro a membro*. La meccanica è essenzialmente la stessa dell'inizializzazione di default membro a membro (costruttore per copia), ma fa uso di un *operatore implicito di assegnamento* per copia al posto del costruttore per copia.
- Se il costruttore per copia di default membro a membro è inadatto per la classe, lo è anche l'*assegnamento di default membro a membro*.

Operatori sovraccaricati: =

- Si può definire un operatore esplicito di *assegnamento per copia*, `operator=()`, con cui si implementa la semantica corretta di copia per la classe. Ad esempio per la classe `ArrayStack`:

```
ArrayStack& ArrayStack::operator=(const ArrayStack &r){
    if(this != &r){
        size=r.size;
        isize=r.isize;
        top=r.top;
        delete [] data;
        data = new char[size];
        for(int i=0;i<=top;i++)
            data[i]=r.data[i];
        cout<<"operator=(const ArrayStack &r)" <<endl;
    }
    return *this;
}
```

Operatori sovraccaricati: =

- Il test condizionale impedisce l'assegnamento di un oggetto a se stesso: è inopportuno nel caso in cui per prima cosa si libera una risorsa associata all'oggetto a sinistra (`delete [] data`) per assegnarvi la corrispondente risorsa (`r.data`) associata allo stesso oggetto a destra dell'assegnamento.

Operatori sovraccaricati: =

```
void TestOperatoreAssegnamento()  
{  
    ArrayStack s1(10);  
    for(int i=0; i<26; i++)  
        s1.Push('a'+i);  
  
    ArrayStack s2;  
    s2=s1;  
}
```

Una possibile uscita

```
ArrayStack() Dimensione iniziale 10  
Raddoppiamento Array --- Nuova dimensione 20  
Raddoppiamento Array --- Nuova dimensione 40  
ArrayStack() Dimensione iniziale 5  
operator=(const ArrayStack &r)  
~ArrayStack()  
~ArrayStack()
```

Operatori sovraccaricati: =

- Analogamente possiamo definire un operatore esplicito di *assegnamento per copia*, `operator=()` per la classe `LinkedList`:

```
LinkedList& LinkedList::operator=(const LinkedList &r){  
    if(this != &r){  
        Clear();  
        for(Node *i=r.first; i!=0; i=i->GetLink())  
            Add(i->GetData());  
        cout<<"operator=(const LinkedList &r)"<<endl;  
    }  
    return *this;  
}
```

Operatori sovraccaricati: =

```
void TestOperatoreAssegnamento()  
{  
    LinkedList l1;  
    l1.Add('a');  
    l1.Add('b');  
    l1.Add('c');  
    l1.Print();  
    LinkedList l2;  
    l2=l1;  
    l2.Print();  
}
```

Una possibile uscita

```
Node(a)  
Node(b)  
Node(c)  
a b c  
Node(a)  
Node(b)  
Node(c)  
operator=(const LinkedList &r)  
a b c  
~Node(a)  
~Node(b)  
~Node(c)  
~Node(a)  
~Node(b)  
~Node(c)
```

Operatori sovraccaricati

Note.

- Se per una classe sono definiti gli operatori `operator+()` e `operator=()`, questi *non* supportano implicitamente l'equivalente operatore composto di assegnamento `+=`. È necessario definire esplicitamente l'operatore sovraccaricato `operator+=()`.
- L'implementazione dell'operatore di assegnamento ha la *semantica del valore*, cioè due oggetti dopo l'assegnamento sono distinti e successive modifiche ad un oggetto non hanno effetto sull'altro.

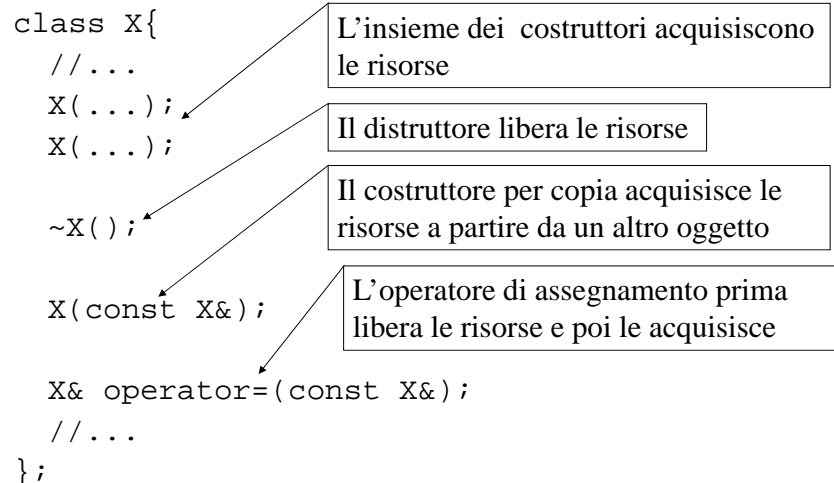
Metodi indispensabili

- Se una classe `x`, tipo definito dall'utente, possiede un costruttore `e`, quindi, un distruttore che eseguono compiti non banali (per esempio *allocare e deallocare memoria dinamica*), deve gestire anche il costruttore per copia e l'operatore di assegnamento. Per evitare che vengano usati quelli di default.
- In generale, una classe che in un qualsiasi metodo acquisisce risorse deve avere un metodo che libera le risorse acquisite: per esempio il metodo `Add()` e `Remove()` della `LinkedList` o `RaddoppiaArray()` e `DimezzaArray()` di `ArrayStack`.

Metodi indispensabili

- In generale, per un tipo `x`, il costruttore per copia `X(const X& o)` si occupa dell'*inizializzazione* per mezzo di un oggetto dello stesso tipo, cioè acquisisce risorse e inizializza.
- L'operatore di assegnamento tra oggetti di tipo `X X& operator=(const X& o)` prima si occupa di liberare le risorse e poi di acquisirle per l'oggetto di sinistra e di iniziarlo.

Metodi indispensabili



Esempio: la classe `FloatArray`

- Come esempio si implementa una classe che *gestisce* l'allocazione dinamica di un array di `float`.
- Usare una classe (*wrapper*) rende l'utilizzo di certe operazioni più "sicuro" e "semplice", per esempio:
 - La deallocazione della memoria è automaticamente eseguita dal distruttore.
 - Si possono inserire dei controlli sul valore degli indici.
 - Si possono fornire funzionalità di stampa.
 - Si possono fornire operazioni implementate con operatori sovraccaricati.
 - Si può fornire un metodo che ritorna la dimensione dell'array.

Esempio: *FloatArray.h*

```
#ifndef FLOATARRAY_H
#define FLOATARRAY_H
#include <iostream>

class FloatArray{
    int size;
    float *data;
public:
    FloatArray(int dim=8, float val=0);
    ~FloatArray();
    FloatArray(const FloatArray &o);
    FloatArray& operator=(const FloatArray &r);

    int Length()const;
    float& operator[](int index);
    float& operator()(int index);
    const FloatArray operator+(const FloatArray &r) const;
    const FloatArray operator+(float x) const;
    friend std::ostream& operator<<(std::ostream &os, const
                                   FloatArray &r);
};
#endif
```

Informatica Medica, I semestre, C++

33

Esempio: *FloatArray.cpp*

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

Nell'esempio si fa uso della macro `assert()` fornita nella Libreria Standard del linguaggio C. Tale macro è utile per verificare una condizione: se la condizione risulta falsa, l'asserzione non è valida e viene stampato un messaggio diagnostico e l'esecuzione del programma si interrompe

```
FloatArray::FloatArray(int dim, float val){
    size=dim;
    data = new float[size];
    for(int i=0;i<size;i++)
        data[i]=val;
    cout<<"FloatArray(" <<size<< ", " <<val<< ")" <<endl;
}
```

Il costruttore acquisisce le risorse e inizializza

```
FloatArray::~FloatArray(){
    delete [] data;
    cout<<"~FloatArray()" <<endl;
}
```

Il distruttore libera le risorse

Informatica Medica, I semestre, C++

34

Esempio: *FloatArray.cpp*

```
FloatArray::FloatArray(const FloatArray &o){
    size=o.size;
    data = new float[size];
    for(int i=0;i<size;i++)
        data[i]=o.data[i];
    cout<<"FloatArray(const FloatArray& o)" <<endl;
}

FloatArray& FloatArray::operator=(const FloatArray &r){
    if(this != &r){
        size=r.size;
        delete [] data;
        data = new float[size];
        for(int i=0;i<size;i++)
            data[i]=r.data[i];
    }
    cout<<"operator=(const FloatArray &r)" <<endl;
    return *this;
}
```

Il costruttore per copia acquisisce le risorse e inizializza

L'operatore di assegnamento prima libera e poi acquisisce risorse e inizializza

Informatica Medica, I semestre, C++

35

Esempio: *FloatArray.cpp*

```
int FloatArray::Length()const{
    return size;
}

float& FloatArray::operator[](int index){
    assert(index>=0 && index<size);
    return data[index];
}

float& FloatArray::operator()(int index){
    assert(index>=0 && index<size);
    return data[index];
}
```

Informatica Medica, I semestre, C++

36

Esempio: operator[]

- In generale, per le classi che rappresentano l'*astrazione di un contenitore*, in cui si possono leggere i singoli elementi, si può definire un operatore di *subscript*: operator[]. Con riferimento alla classe FloatArray i metodi Get() e Set() possono essere sostituiti con operator[].
- Tale operatore deve poter comparire sia sul *lato sinistro* sia su quello *destro* di un operatore di assegnamento.

Esempio: operator[]

- Quindi il suo valore di ritorno deve essere un *lvalue*. Ciò si ottiene specificando il tipo di ritorno come un *riferimento*.
- L'uso del *reference* permette di utilizzare l'operatore di *subscript* in un modo *intuitivo*, come per i tipi predefiniti:

```
int a[3]; ...; a[1]=23;
```
- La stessa funzionalità si può implementare con l'overloading dell'operatore di chiamata a funzione, operator().

Esempio: FloatArray.cpp

```
const FloatArray FloatArray::operator+(const FloatArray &r)
const{
    assert(size==r.size);
    FloatArray tmp(size);
    for(int i=0;i<size ;i++)
        tmp.data[i]=data[i] + r.data[i];
    return tmp;
}
```

Operatore di somma tra due array

Operatore di somma tra un array
e un valore float

```
const FloatArray FloatArray::operator+(float x) const{
    FloatArray tmp(size);
    for(int i=0;i<size ;i++)
        tmp.data[i]=data[i] + x;
    return tmp;
}
```

Esempio: FloatArray.cpp

```
ostream& operator<<(ostream &os, const FloatArray &r) {
    for(int i=0;i<r.size ;i++)
        os<<r.data[i]<<" ";
    os<<endl;
    return os;
}
```

Operatore di stampa per un oggetto
della classe: è una funzione friend,
non un metodo della classe

Esempio: *test.cpp*

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

void test1();

int main(){
    test1();
    return 0;
}

void test1(){
    FloatArray a, b(5,3);
    cout<<a<<b;
}
```

Una possibile uscita

```
FloatArray(8,0)
FloatArray(5,3)
0 0 0 0 0 0
3 3 3 3 3
~FloatArray()
~FloatArray()
```

Gli oggetti sono locali, ma i dati sono allocati dinamicamente.

Operatore di stampa per gli oggetti.

Esempio: *test.cpp*

```
void test2(){
    FloatArray a(10,2), b(10);
    for(int i=0;i<b.Length();i++)
        b(i)=i;
    cout<<a+b;
}
```

Oggetto funzione

Costruttore di tmp

Costruttore per copia dovuto al return tmp;

Una possibile uscita

```
FloatArray(10,2)
FloatArray(10,0)
FloatArray(10,0)
FloatArray(const FloatArray& o)
~FloatArray()
2 3 4 5 6 7 8 9 10 11
~FloatArray()
~FloatArray()
~FloatArray()
```

Esempio: *test.cpp*

```
void test3(){
    FloatArray a(10,2), b;

    for(int i=0;i<a.Length();i++)
        a[i]=i;
    b=a+3.3;
    cout<<b;
}
```

Una possibile uscita

```
FloatArray(10,2)
FloatArray(8,0)
FloatArray(10,0)
FloatArray(const FloatArray& o)
~FloatArray()
operator=(const FloatArray &r)
~FloatArray()
3.3 4.3 5.3 6.3 7.3 8.3 9.3 10.3 11.3 12.3
~FloatArray()
~FloatArray()
```

operator[]

Somma tra oggetto e float

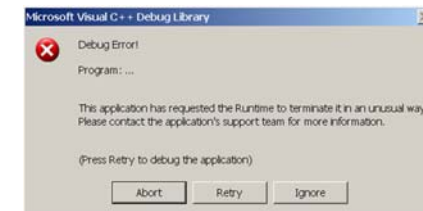
Assegnamento tra oggetti

Esempio: *test.cpp*

```
void test4(){
    FloatArray a(8,2);
    a(20)=3;
}
```

Una possibile uscita

```
FloatArray(8,2)
Assertion failed: index>=0 && index<size, file g:\users\fabio\insegnamenti\informatica_medica\esempi\lezione10\floatarray.cpp, line 44
```



Esempio: *test.cpp*

```
void test5(){  
    FloatArray a(5,2), b(5,7);  
  
    a=b;  
    b(0)=0;  
    cout<<a<<b;  
}
```

Se si assegna b ad a, una modifica su b non influenza a.

Una possibile uscita

```
FloatArray(5,2)  
FloatArray(5,7)  
operator=(const FloatArray &r)  
77777  
07777  
~FloatArray()  
~FloatArray()
```