Ancora sul progetto di Classi

1. attributi e metodi di classe in C++ e Java
2. overloading operatori
3. funzioni “friend”
4. metodi **const**

} Specifici del C++

Ancora sul progetto di Classi

1. attributi e metodi di classe in C++ e Java
2. overloading operatori
3. funzioni “friend”
4. metodi **const**

attributi e metodi di classe: *richiami*

- Oltre a **attributi** e **metodi di istanza** esistono anche **attributi** e **metodi di classe**.

- Le operazioni e gli attributi di classe sono necessari per far fronte alle situazioni che non possono essere responsabilità di un qualunque oggetto singolo.

- Esempi per la classe **`Robot`** :
  - Metodo di classe: **`New()`**
  - Attributo di classe: **`numeroRobotCreati`**

attributi e metodi di classe: *C++*

Come viene inplementato in C++ il meccanismo di classe?

```
class Employee {
public:
  Employee( string,  string );  // costruttore
  ~Employee();                  // distruttore
  string getFirstName();
  string getLastName();
  static int getCount();  // metodo di classe; ritorna il
                          // numero di oggetti instanziati
private:
        string firstName;
        string lastName;
  static int count;  // attributo di classe; mantiene il numero di
                     // oggetti instanziati
};
```

attributi e metodi di classe: *C++*

```
...
#include “Employee.h”
...
int main()
{
...
   cout << "Numero di oggetti Employee istanziati: "
        << Employee::getCount() << endl << endl;
...

};
```

In C++, per inviare un messaggio ad un oggetto si usa l’operatore punto (.)

In C++, per inviare un messaggio ad una classe si usa l’operatore di scope (::)

Dato che è possibile chiedere alla classe il numero di oggetti istanziati come prima operazione di un main, quando viene stabilito il valore iniziale di un attributo di classe?

attributi e metodi di classe: *C++*

```
// employee.cpp
#include "Employee.h"

// Inizializzazione dell'attributo di classe
int Employee::count = 0;

// Implementazione del metodo di classe
int Employee::getCount() { return count; }

// Costruttore
Employee::Employee( const string first, const string last )
{ ...
```

Il valore iniziale di un attributo di classe è di solito specificato durante l’implementazione dei metodi.

attributi e metodi di classe: *C++*

```
// Costruttore
Employee::Employee( string first, string last ) {
  firstName = first;
  lastName =  last;
  count++;  // incrementa il numero di oggetti istanziati
}

// Distruttore
Employee::~Employee() {
   count--;  // decrementa il numero di oggetti istanziati
}
```

In questo specifico esempio, le modifiche all'attributo di classe count sono a carico sia del costruttore che del distruttore

attributi e metodi di classe: *Java*

```
public class Employee {
   private String firstName;
   private String lastName;
   private static int count = 0;  // numero di oggetti in memoria

   public Employee( String first, String last )
   { firstName = first;
     lastName = last;
     ++count;  // incrementa il conteggio di classe degli impiegati
   }
   protected void finalize() { --count;  } // decrementa il numero di impiegati

   public String getFirstName()  {  return firstName;  }
   public String getLastName()  {  return lastName;  }

   public static int getCount() { return count;  }  // metodo di classe

} // end class Employee
```

Valore inizale dell'attributo di classe inizializzato nella definizione della classe

attributi e metodi di classe: *Java*

In Java, per inviare un messaggio ad un **classe** si usa sempre l'operatore punto (.)

```
public class EmployeeTest {

   public static void main( String args[] )
   {
      System.out.print("Impiegati iniziali: " + Employee.getCount());  //0

      // crea 2 impiegati: count dovrebbe essere 2
      Employee e1 = new Employee( "Susan", "Baker" );
      Employee e2 = new Employee( "Bob", "Jones" );
      System.out.print("Impiegati attuali: " + Employee.getCount());  //2

      e1 = null;  e2 = null;
      System.gc(); // suggerisce di chiamare il garbage collector

      System.out.print("Impiegati finali: " + Employee.getCount());  //0

} // end class EmployeeTest
```


attributi e metodi di classe: *altri esempi*

- Progettare una classe Utente che permette di creare oggetti che modellano i dipendenti di un certo ente.
- Ogni utente è caratterizzato da Nome, Cognome, Stipendio e Matricola
- La matricola deve essere unica per ogni utente
- In fase di creazione di un oggetto utente, è responsabilità della classe assegnare univocamente la matricola.
- Discutere possibili soluzioni e analizzare il codice proposto ***(esempio AssignUniqueKey)***
- Analizzare il codice Java della classe Employee proposto nell'esempio ***Employee***

Ancora sul progetto di Classi

1. attributi e metodi di classe
2. overloading operatori
3. funzioni “friend”
4. metodi **const**

Overloading operatore di somma

Sostituendo

```
Pair Pair::sum(const Pair & c) {
  Pair ris;
  ris.first = this->first + c.first;
  ris.second = this->second + c.second;
  return ris;
}
```

con:

```
Pair Pair::operator+(const Pair & c) {
  Pair ris;
  ris.first = this->first + c.first;
  ris.second = this->second + c.second;
  return ris;
}
```

Overloading operatore di somma

È possibile sostituire la forma poco comune (ma coerente con l'OOP):

```
int main()
{
   Pair x(-3,4), y(2), z;
   z = x.sum(y);
   cout << z.getFirst() << z.getSecond();
   ...
}
```

con:

```
int main()
{
   Pair x(-3,4), y(2), z;
   z = x + y;      // corrisponde a z=x.operator+(y)
   cout << z.getFirst() << z.getSecond();
   ...
}
```


Esercizi

- Modificare la classe Fraction sostituendo sum, sub, mul, e div con operator+, operator-, operator* e operator/ . Aggiungere gli operatori =, ==, <, > e altri operatori come +=, <=, etc.

- Vedere gli esempi ulteriori:
  - *MyListGood* (per la ridefinizione dell'operatore =)
  - *overloadingIncrement* (Vedere cap. 11 libro di testo)
  - *overloadingCinCout* (studiare prima le funzioni friend!)

Ancora sul progetto di Classi

1. attributi e metodi di classe
2. overloading operatori
3. funzioni "friend"
4. metodi **const**

Overloading: *problema*

- Discutere il seguente problema: come è possibile ridefinire l'operatore << in modo da permettere quanto segue?

```
int main()
{
   Pair x(-3,4), y(2), z;
   z = x + y;
   cout << z;
   ...
}
```

Overloading: *soluzione 1*

```
ostream & operator<<( ostream &out, const Pair & p )
{
    out << "(" << p.getFirst() << "," << p.getSecond() << ")";
    return out;   // permette cout << p1 << p2 << p3;
}

int main()
{
   Pair x(-3,4), y(2), z;
   z = x.sum(y);
   cout << z;
   ...
}
```

Attenzione: qui operator<<() non è un metodo di qualche classe, ma solo una funzione esterna al main

Problema: ma se non esistessero i metodi get per tutti gli attributi dell'oggetto Pair?

Overloading: *soluzione 2 – nel file .h*

Nel caso non esistano i metodi get per tutti gli attributi è necessario ricorrere alle funzioni friend !!

```
class Pair {
  friend ostream & operator<<( ostream &output, const Pair & p )
  private:
     int first, second;
  public:
     Pair ();
     Pair ( int a );
     Pair ( int a, int b );
     Pair operator+(const Pair & c);
};
```

Attenzione: qui non dispongo dei metodi get per tutti gli attributi

Overloading: *soluzione 2 – nel file .cpp*

```
// Pair.cpp - Implementazione interfaccia classe Pair
#include "pair.h“

// notare che la funzione friend non ha lo scope di classe perché
// NON E’ un metodo della classe Pair
ostream & operator<<( ostream &out, const Pair & p ) {
    out << "(" << p.first << "," << p.second << ")";
    return out;   // permette cout << p1 << p2 << p3;
}

Pair::Pair () {
  first = second = 0;
}

Pair::Pair ( int a ) {
  first = second = a;
}
...
```

La funzione può accedere direttamente agli attributi di Pair perché ne è friend

Funzioni friend: *alcune info*

- Una funzione friend di una classe è una funzione esterna alla classe, ma che ha il diritto di accedere ai membri privati di essa
- Att.ne: le funzioni friend non sono metodi di una classe
- Limitare l’uso delle funzioni friend, perché rompono l’incapsulamento
- Sono necessarie per l’overloading degli operatori >> e << nel caso in cui una classe non abbia tutti i metodi di lettura (metodi get)
- Nel caso una classe abbia tutti i metodi get, si mette il prototipo di operator<< in .h (ma fuori dalla classe) e la sua implementazione in .cpp

Ancora sul progetto di Classi

1. attributi e metodi di classe
2. overloading operatori
3. funzioni “friend”
4. metodi **const**

Osservazione

```
ostream & operator<<( ostream &out, const Pair & p )
{
   out << “(“ << p.getFirst() << “,” << p.getSecond() << “)”;
   return out;   // permette cout << p1 << p2 << p3;
}

int main()
{
   Pair x(-3,4), y(2), z;
   z = x.sum(y);
   cout << z;
   ...
}
```

Attenzione: qui si ha un errore di compilazione perché p è const in operator<< mentre il metodo getFirst() può (anche se non lo fa effettivamente) modificare le strutture dati dell’oggetto di invocazione

Come risolvere il problema?

Prima soluzione

```
ostream & operator<<( ostream &out, Pair & p )
{
    out << "(" << p.getFirst() << "," << p.getSecond() << ")";
    return out;   // permette cout << p1 << p2 << p3;
}

int main()
{
   Pair x(-3,4), y(2), z;
   z = x.sum(y);
   cout << z;
   ...
}
```

Prima soluzione: passare p a operatore<< come riferimento NON costante.

Pessima soluzione.

Seconda soluzione

```
class Pair {
  private:
    int first, second;
  public:
    Pair ();
    Pair ( int a );
    Pair ( int a, int b );
    int getFirst() const;
    int getSecond() const;
    Pair operator+(const Pair & c);
};
```

Seconda soluzione: definire getFirst() e getSecond() come metodi const.

Questo implica che i due metodi non possono modificare l'oggetto di invocazione.

Buona soluzione (da usare in modo massiccio).

Esempi

- Vedere l'esempio nella cartella *attributiMetodiConst* (codice allegato alle presenti slide)
- L'esempio mostra anche altri casi sull'uso del **const**
- …