Overloading di Operatore

Perche'

- In C++ si creano e manipolano dei tipi di dato creati dall utente (il programmatore)
- Il C++ mette a disposizione degli strumenti per creare e distruggere gli oggetti creati con tipi definiti dall'utente
- Sarebbe utile poter estendere questo parallelismo e poter operare su i tipi definiti dall'utente così come si opera su i tipi primitivi (int, char, etc)
- Questo permetterebbe la manipolazione degli oggetti con un formalismo intuitivo e conciso: ad esempio effettuare la somma fra le celle di due vettori a e b scrivendo semplicemente a+b

Operatore o funzione?

- Il C++ possiede gia' il meccanismo di overload (sovraccaricamento) per le funzioni
- Gli operatori non sono altro che un modo conciso per scrivere delle funzioni
- Ad esempio `a+b` è in realtà `funzione_somma(a,b)`
- In C++ gli operatori si possono indicare esplicitamente e sovraccaricarli, rendendoli così specifici per i tipi utente
- **Nota**: non è possibile creare operatori nuovi, cioè utilizzare un simbolo non previsto come un operatore, come $ o £ o ¬

Operatori ammissibili in C++

- Operatori per i quali è ammesso l'overload:

```
+   -   *   /   %   ^   &   |   ~ !   =   <   >   ,

[ ]  ( )  new  delete
+=   -=   *=   /=   %=   ^=   &=   |=   <<   >>
>>=  <<=   ==   !=   <=   >=   &&   ||   ++   --

->   ->*
```

- Operatori per i quali non è ammesso l'overload:

```
.   ::   .*   ?: sizeof
```

Overloading degli operatori

- L'operazione di definire un comportamento per un operatore va sotto il nome di overloading dell'operatore (sovraccarico dell'operatore)
- Per effettuare l'overloading di operatore si procede *come* nel caso di funzioni (membro o non)
- Il nome della funzione è composto dalla parola chiave *operator* seguita dal simbolo dell'operatore
- Ad esempio per sovraccaricare il + si ha *operator+*

```
MyClass MyClass::operator+(const MyClass & obj){
    //definizione operazione
}
```

Operatori predefiniti

- Perché un operatore possa operare su gli oggetti di una classe definita dall'utente deve *necessariamente* essere ridefinito
- ..cioe' non esiste un meccanismo automatico per creare ad esempio una funzione di somma per tipi definiti dall'utente
- Fanno eccezione:
  - Operatore di assegnamento =

    il suo comportamento di default è di eseguire una copia di membro a membro dei dati di una classe
  - Operatore di indirizzo &

    il suo comportamento di default è di restituire l'indirizzo di memoria dell'oggetto
- **Tuttavia** anche = e & possono essere ridefiniti se serve

Operatori derivati

- Anche gli operatori derivati devono essere esplicitamente sovraccaricati e definiti
- Non ci si deve aspettare che se si è implementato l'overload per + automaticamente sia disponibile +=

- Per assicurare consistenza si dovrebbe sempre riutilizzare un operatore per ridefinire il comportamento degli altri operatori simili
  - usare + per definire +=
  - usare == per definire !=

La semantica di un operatore

- Il senso (semantica) della ridefinizione di un operatore dovrebbe essere quanto più vicino al senso che l'operatore ha per i tipi predefiniti

- Anche se è possibile definire in modo arbitrario ciò che fa un operatore si crea solo confusione se un operatore ha una semantica inaspettata

- Ad esempio si può sovraccaricare l'operatore + per eseguire una sottrazione (!?) ma il codice diventa fuorviante e di difficile interpretazione

Arita' di operatore

- Non si può ridefinire un operatore che opera su i tipi predefiniti (int, char, float, etc)
- Non si può cambiare l'arità di un operatore né la sua associatività
- La arità è il numero di parametri che l'operatore prende
  - a+b ha arità 2 infatti è operator+(int a, int b)
  - a++ ha arità 1 infatti è operator++(int &a)

Operatore unario

- Un operatore unario può essere ridefinito come funzione membro senza argomenti
- ..oppure come funzione non membro con un argomento

Overloading di operatore unario

- L'argomento di un operatore unario deve essere:
  - un oggetto di una classe
  - un riferimento ad un oggetto di una classe
- Il passaggio di un riferimento è *necessario* nel caso di *funzioni non membro* quando l'operatore deve modificare l'oggetto che invoca l'operatore stesso
- Esempio
  - ridefinendo l'operatore ++ si ha che x++ viene trattato come x.operator++( )
  - non è necessario per l'operatore !

Overloading di operatore unario

```
class MyInt{
   friend MyInt& operator--(MyInt &);
   public:
      MyInt(int usr_dat=0){dat=usr_dat;}
      MyInt& operator++(){
        dat++;
        return *this;
      }
      bool operator!()const{
        if(dat==0) return true;
        else return false;
      }
    private:
      int dat;
    };

    MyInt& operator--(MyInt & obj){
        obj.dat--;return obj;
    }
```

Overloading di operatore binario

- Un operatore binario può essere ridefinito come funzione membro con un argomento
- ..oppure come funzione non membro con due argomenti

- L'argomento deve essere un oggetto o un riferimento ad un oggetto di una classe
- Ad esempio ridefinendo l'operatore + si ha che x+y viene trattato come x.operator+(y)

Overloading di operatore binario

```
class MyInt{
   friend MyInt operator-=(MyInt &, const MyInt &);
    public:
      MyInt(int usr_dat=0){dat=usr_dat;}
      MyInt operator+(const MyInt &obj){
        MyInt temp;
        temp.dat=(*this).dat+obj.dat;
        return temp;
      }
      MyInt operator+=(const MyInt &obj){
        (*this).dat += obj.dat; //oppure dat += obj.dat;
        return *this;
      }
    private:
      int dat;
};

MyInt operator-=(MyInt & a, const MyInt & b){
       a.dat-=b.dat;
       return a;
}
```

Cosa restituiscono gli operatori

- In linea di principio si può restituire:
  - copia dell'oggetto risultato
  - riferimento all'oggetto risultato
- Es:

```
MyInt operator++(){
  (*this).dat++;
  return *this;
}
MyInt& operator++(){
  (*this).dat++;
  return *this;
}
```

- Dipende dalla semantica dell'operatore

Mai un alias ad un oggetto temporaneo

- Attenzione! MAI restituire un alias ad un oggetto temporaneo (locale)!

```
MyInt & operator-=(MyInt & a, const MyInt & b){
    MyInt temp;
    temp.dat=a.dat-b.dat;
    return temp;
  }
```

- E' un errore!
- Una espressione come c=d+(a-b) potrebbe avere un risultato imprevedibile
- Nessuno garantisce infatti che la cella di memoria dove risiede l'oggetto temp non sia sovrascritta prima di eseguire la somma!

Operatori come funzioni `friend`

- Può capitare che sia utile che un operatore sia una funzione non-membro di una classe
- Es. se si è definito l'operatore + per una classe ObjClass allora:

```
Obj+100; //OK
```

- tuttavia:

```
100+Obj; //ERROR
```

- Infatti nel secondo caso stiamo utilizzando un operatore del tipo predefinito **`int`**
- Cioè è l'oggetto 100 a chiamare l'operatore + e non Obj

Operatori come funzioni **friend**

```
class MyInt{
    friend MyInt operator+(const MyInt &obj1, int);
    friend MyInt operator+(int, const MyInt &obj1);
    public:
      MyInt(int usr_dat=0){dat=usr_dat;}
    private:
      int dat;
};
MyInt operator+(const MyInt &obj1, int x){
       MyInt temp;
       temp.dat = obj1.dat + x;
       return temp;
}
MyInt operator+(int x, const MyInt &obj1){
       MyInt temp;
       temp.dat = x + obj1.dat;
       return temp;
}
```

Overloading di operatore di un int?

- Non stiamo ridefinendo l'operatore + per gli interi con la seguente espressione:

```
friend MyInt operator+(int, const MyInt &obj1);
```

- Infatti la sola ridefinizione del + non permessa è:

```
int operator+(int, int);
```