Sovraccaricamento degli Operatori

Operatori

- simboli convenzionali che rendono più semplice la programmazione
- (a+b)*c più facile di Moltiplica(Somma(a,b),c)
- E' fondamentale che siano concisi
- C++ supporta operazioni per i tipi nativi
- Spesso però noi usiamo tipi astratti

Overaload degli operatori

- Il programmatore può creare nuove funzioni, ridefinendo i simboli delle operazioni
- Ciò le rende applicabili anche ai tipi astratti
- A differenza delle normali funzioni, usiamo solo il simbolo dell'operatore

Es

```
Punto p1, p2, p;
p=p1+p1;
```

Sintassi

- A parte qualche caso, possono restituire qualunque tipo di dato
- Non si può cambiare:
  - precedenza, raggruppamento e numero degli operandi.
- Tipicamente non possono avere argomenti per default.

Sintassi

operazione: p1+p2

```
punto operator+(const punto& p1, const punto& p2)
{  punto temp(0.0,0.0);
   temp.x=p1.x+p2.x;
   temp.y=p1.y+p2.y;
   return p.temp
}
```

- Il passaggio per riferimento serve solo per efficienza.

Metodi della classe vs funzioni esterne

- Se accedono a membri privati della classe, possono appartenere soltanto a una delle seguenti tre categorie:
  1. sono metodi pubblici non statici della classe;
  2. sono metodi pubblici statici della classe;
  3. sono funzioni friend della classe.
- 2 non conviene
  - chiamare l'operatore implicherebbe qualifircarlo con il nome della classe punto::operator+(p1,p2)

In generale

- Conviene il metodo 1 per funzioni unarie
- Esso è obbligatorio se il primo operando è un oggetto della classe e l'operatore lo restituisce come valore sinistro (dell'operatore stesso)
  - Es: =, +=, etc.
- In tal caso, nella definizione del metodo il num. degli argomenti deve essere ridotto di un'unità rispetto al numero di operandi;
  - se l'operatore è unario, la funzione non deve avere argomenti
- se il risultato dell'operazione è l'oggetto stesso, l'istruzione di ritorno deve essere:

```
return *this;
```

Esempio

operazione : p += p1 ;

definizione metodo :

```
punto& punto::operator+=(const punto& p1)
{
            x += p1.x ;
            y += p1.y ;
            return *this ;
}
```

Operatori di flusso

- creare un (ulteriore) overload di << (>>)

es:

`cout << a` (a istanza di una classe A)

- Dobbiamo sapere che:
  - cout, oggetto globale generato all'inizio dell'esecuzione del programma, é un'istanza della classe `ostream`, che viene detta "classe di flusso di output"
  - il primo argomento dovrà essere lo stesso oggetto cout (in quanto é il left-operand dell'operazione),
  - il secondo argomento, (right-operand) dovrà essere l'oggetto da trasferire in output.
  - la funzione dovrà restituire by-reference lo stesso primo argomento (cioè sempre cout), per permettere l'associazione di ulteriori operazioni nella stessa istruzione.

Funzione per l'overaload di <<

```
ostream& operator<<(ostream& out, const A& a)
{
 ........ out << a.ma;
 ........ return out
}
```

- (`ma` è un membro di `A` di tipo nativo)
- Si noti che
  - Il primo argomento appartiene a `ostream`
  - L'output è restituito per referenza (perché è il left-value)

Operatori Binari

- Hanno almeno un operando che è oggetto della classe, non importa se left o right
  - per gli operatori in notazione compatta l'oggetto della classe deve essere sempre left.
- L'altro operando può essere un oggetto di qualsiasi altro tipo, nativo o astratto.
- Possono esistere parecchi overload dello stesso operatore, ciascuno con un operando di tipo diverso.
- Se si vuole salvaguardare la proprietà commutativa di certe operazioni (+ * & | ^ == != && ||), o la simmetria di altre (< con >= e > con <=), occorrono, per ognuna di esse, due funzioni.

Operatori Binari

- il numero di funzioni può essere minimizzato utilizzando i costruttori con un argomento,
- Questi definiscono una conversione implicita di tipo:
  - tutti i tipi coinvolti nelle operazioni sono convertiti implicitamente nel tipo della classe
  - ogni operazione è perciò implementata da una sola funzione, quella che opera su due oggetti della stessa classe.

Esempio

Somma fra un punto p e un double s (s viene sommato a ogni componente di p).

Costruttore:

```
punto::punto(double d):x(d),y(d)   { }
```

`p + s` **e** `s + p`

- comportano conversione implicita di `s` da `double` a `punto`.