

Regole per l'overloading degli operatori

RULES

1. YOU CAN....

2. YOU CAN'T...

1) Non si possono cambiare:

- posizione (prefissa/infissa/postfissa)
- numero operandi
- precedenze e associatività

2) Tra gli argomenti deve essere presente almeno un tipo definito dall'utente

3) Gli operatori "=", "[]" e "->" si possono sovraccaricare solo come metodi (interni)

4) **Non si possono** sovraccaricare gli operatori ".", "::", "sizeof", "typeid", i cast e l'operatore condizionale ternario "? :"

5) Gli operatori "=", "&" e "," hanno una versione standard

Operatore condizionale ternario

```
booleanExpr ? expr1 : expr2;
```

```
// ESEMPI
```

```
orario oraApertura = (day == SUNDAY) ? 15 : 9;
```

```
int max(int x, int y) {  
    return x > y ? x : y;  
}
```

Operatore virgola

L'operatore virgola separa delle espressioni (i suoi argomenti) e ritorna il valore solamente dell'ultima espressione a destra, mentre tutte le altre espressioni sono valutate per i loro effetti collaterali.

Why?


```
int main() {  
    int a = 0, b = 1, c = 2, d = 3, e = 4;  
    a = (b++, c++, d++, e++);  
    cout << "a = " << a << endl; // stampa 4  
    cout << "c = " << c << endl; // stampa 3  
}
```


Esercizio



Scrivere una definizione degli operatori "-",
"==", ">" e "<" per la classe **orario**. Scrivere
una definizione degli operatori "+", "-", "==",
">" e "<" per la classe **complesso**.



 **C&operator=(const C&)**


```
int x=1, y=4, z=7;
```

```
x=y=z; // ←
```

```
cout << x << " " << y << " " << z; // cosa stampa?
```

```
7 7 7 // valutazione "right to left"
```




 **C&operator=(const C&)**


```
int x=1, y=4, z=7;
```

```
(x=y)=z; // 
```

```
cout << x << " " << y << " " << z; // cosa stampa?
```

```
7 4 7 // valutazione "left to right"
```

Assignment

 **C&operator=(const C&)**

```
int x=1, y=4, z=7;
```

```
x=y=z; // ←
```

```
cout << x << " " << y << " " << z; // cosa stampa?
```

```
int& f(int& a) {a=3; return a;}
```

```
int x=5, y=8;
```

```
f(y=x); // ←
```

```
cout << x << " " << y; // cosa stampa?
```


Costruttore di copia

Il costruttore di copia **C (const C&)** viene **invocato automaticamente** nei seguenti tre casi:

- 1 - quando un oggetto viene dichiarato ed inizializzato con un altro oggetto della stessa classe, come nei seguenti due casi:

```
orario adesso(14,30);  
orario copia = adesso; // inizializza copia  
orario copia1(adesso); // inizializza copia1
```


- 2 - quando un oggetto viene **passato per valore** come parametro di una funzione, come in:

```
ora = ora.Somma(DUE_ORE_E_UN_QUARTO);
```

dove `Somma` è la funzione:

```
orario orario::Somma(orario o) const {  
    orario aux;  
    aux.sec = (sec + o.sec) % 86400;  
    return aux;  
}
```

- 3 - quando una funzione **ritorna per valore** tramite l'istruzione `return` un oggetto, come in

```
return aux;
```

nella funzione precedente.

Attenzione al caso 3

Il compilatore g++ di **default** compie la seguente ottimizzazione:



*"The C++ standard allows an implementation **to omit creating a temporary which is only used to initialize another object of the same type**. Specifying this option disables that optimization, and forces g++ to call the copy constructor in all cases."*

L'ottimizzazione è sottile: dice che ogni qualvolta si dovrebbe creare un oggetto temporaneo "inutile" usato solamente per inizializzare un nuovo oggetto dello stesso tipo, il temporaneo non viene creato.

La specifica di tale ottimizzazione non è completamente deterministica e quindi **a fini didattici** spesso non considereremo tale ottimizzazione.

Intenderemo quindi sempre l'opzione
g++ -fno-elide-constructors
che disabilita questa ottimizzazione nel compilatore.



```
C fun(C a) {return a;}
```

```
int main() {
```

```
    C c;
```

```
    fun(c);
```

```
    // 2 invocazioni del costr. di copia
```

```
    C y = fun(c);
```

```
    // g++ => 2 sole invocazioni del costr. di copia e non 3!
```

```
    C z; z = fun(c);
```

```
    // 2 invocazioni del costr. di copia
```

```
    fun(fun(c));
```

```
    // g++ => 3 sole invocazioni del costr. di copia e non 4!
```

```
}
```

Esercizio: Cosa stampa?



```
#include <iostream>
using namespace std;

class Puntatore {
public:
    int* punt;
};

int main() {
    Puntatore x, y;
    x.punt = new int(400);
    y = x;
    cout << "*" (y.punt) = " << * (y.punt) << endl;
    * (y.punt) = 100;
    cout << "*" (x.punt) = " << * (x.punt) << endl;
}
```


Esercizio cavillo: Cosa stampa?



```
#include <iostream>
using namespace std;

class Vettore {
public:
    int vett[10];
};

int main() {
    Vettore x, y;
    x.vett[2] = 200;
    y = x;
    cout << "y.vett[2] = " << y.vett[2] << endl;
    y.vett[2] = 0;
    cout << "x.vett[2] = " << x.vett[2] << endl;
}
```

Esercizio cavillo: Digressione su array



```
int main() {  
    int a[10];  
    int b[10];  
    b=a; // ILLEGALE: array type int[10] is not assignable  
}
```



```

#include <iostream>
using namespace std;

class Vettore {
public:
    int vett[10];
};

int main() {
    Vettore x, y;
    x.vett[2] = 200;
    y = x;
    cout << "y.vett[2] = " << y.vett[2] << endl;
    y.vett[2] = 0;
    cout << "x.vett[2] = " << x.vett[2] << endl;
}

```

Spiegazione (dal documento dello standard C++03, paragrafo 12.8/13)

The implicitly-defined assignment operator for class X performs memberwise assignment of its subobjects. ... Each subobject is assigned in the manner appropriate to its type:

...

-- if the subobject is an array, **each element is assigned**, in the manner appropriate to the element type

...

Funzionalità di stampa per oggetti orario



L'operatore di output del C++ è "<<".
Facciamo overloading di questo operatore.

```
std::ostream& orario::operator<<(std::ostream& os) const  
{  
    return os << Ore() << ':' << Minuti()  
              << ':' << Secondi();  
}
```

Per poter invocare "a cascata" l'operatore <<
come al solito occorre che esso ritorni lo stream
per riferimento.


```
orario le_tre(15,0);  
orario le_quattro(16,0);  
le_quattro << ((le_tre << cout)  
               << " vengono prima delle ");
```



Spiegazione: si tratta delle seguenti chiamate

```
le_quattro.operator<<(cout);  
le_tre.operator<<(cout);
```

Vorremmo invece, come al solito:

```
cout << le_tre << " vengono prima delle " << le_quattro;
```


Overloading di << come **funzione esterna** alla classe:

```
ostream& operator<<(ostream& os,const orario& o) {  
    return os << o.Ore() << ':' << o.Minuti()  
        << ':' << o.Secondi();  
}
```

```
orario le_tre(15,0), dodici(12,0);  
  
cout << "Adesso sono le " << le_tre << endl;  
// stampa: Adesso sono le 15:0:0  
  
cout << "Fra dodici ore saranno le "  
    << le_tre + dodici << endl;  
// stampa: Fra dodici ore saranno le 3:0:0
```