

Overloading di operatori ed ereditarietà in C++

Emanuele Ing. Benatti

Perchè L'overloading di operatori.

Il **C++** supporta, come ogni altro linguaggio, un insieme di **operazioni** per i suoi **tipi nativi**. Tuttavia la maggior parte dei concetti utilizzati comunemente non sono facilmente rappresentabili per mezzo di **tipi nativi**, e bisogna spesso fare ricorso ai **tipi astratti**.

Nel linguaggio C++ il tipo astratti fondamentale che si usa è il tipo **classe**.

```
class Point
{
    int x;
    int y;
}
```

Overloading di operatori

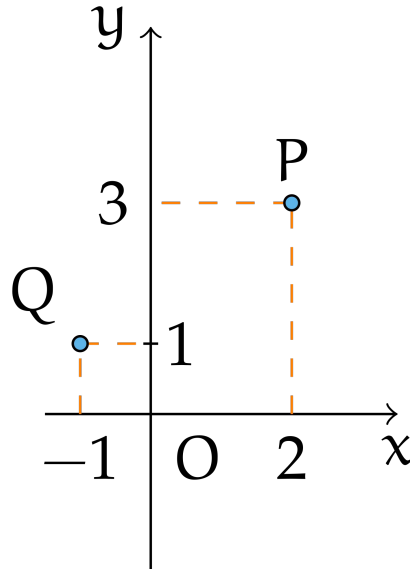
In una classe per overloading intendiamo la “**ridefinizione**” di un **operatore** che può essere applicato ad una **classe ed ai suoi oggetti**.

Domanda: che senso ha, ad esempio, ridefinire l'operatore più per la classe Point?

Overloading di operatori

Domanda: che senso ha, ad esempio, ridefinire l'operatore più per la classe Point?

Ad esempio la possibilità di calcolare il punto dato dalla somma di due punti.



Overloading di operatori: l'operatore +

Per ottenere l'**overload** di un **operatore** bisogna creare una **funzione** il cui **nome** (che eccezionalmente non segue le regole generali dei **metodi di classe**) deve essere costituito dalla **parola-chiave** **operator** seguita, dal simbolo dell'**operatore** (es.: **operator+**). Gli **argomenti** della **funzione** devono corrispondere agli **eventuali operandi** dell'**operatore**. In questo caso **operator +** è una funzione della classe **Point**

```
operazione : p = p1+p2 ;  
funzione somma :  
point operator+(const point& p2)  
{  
    point ptemp(0.0,0.0);  
    ptemp.x = this->x+ p2.x ;  
    ptemp.y = this->y + p2.y ;  
    return ptemp ;  
}
```


Passaggio by
reference

p2 e this sono i due
parametri della
somma

Overloading di operatori: La parola chiave const

La parola chiave const viene usata nella firma del metodo perchè non devo essere in grado di modificare, in nessun modo i parametri operandi presenti.

```
operazione : p = p1+p2 ;  
funzione somma :  
point operator+(const point& p2)  
{  
    point ptemp(0.0,0.0);  
    ptemp.x = this->x+ p2.x ;  
    ptemp.y = this->.y + p2.y ;  
    return ptemp ;  
}
```



Costruisco pertanto un nuovo oggetto tipo point che rappresenta la somma dei due oggetti, e questo oggetto sarà restituito alla funzione chiamante.

Overloading di operatori: l'operatore + , definito fuori dalla classe

Per ottenere l'**overload** di un **operatore** fuori dalla classe Point bisogna creare una **funzione** (friend) il cui **nome** (che eccezionalmente non segue le regole generali dei **metodi di classe**) deve essere costituito dalla **parola-chiave operator** seguita, dal simbolo dell'**operatore** (es.: **operator+**). Gli **argomenti** della **funzione** devono corrispondere agli **eventuali operandi** dell'**operatore**.

```
operazione : p = p1+p2 ;  
funzione somma :  
point operator+(const point& p1, const point& p2)  
{  
    point ptemp(0.0,0.0);  
    ptemp.x = p1.x + p2.x ;  
    ptemp.y = p1.y + p2.y ;  
    return ptemp ;  
}
```

Passaggio by
reference

p1 e p2 sono i due
parametri della
somma

Overloading di operatori: l'operatore +

Anche in questo caso costruisco pertanto un nuovo oggetto tipo point che rappresenta la somma dei due oggetti, e questo oggetto sarà restituito alla funzione chiamante.

Oggetto risultato

```
operazione : p = p1+p2 ;  
funzione somma :  
point operator+(const point& p1, const point& p2)  
{  
    point ptemp(0.0,0.0);  
    ptemp.x = p1.x + p2.x ;  
    ptemp.y = p1.y + p2.y ;  
    return ptemp ;  
}
```


Overloading di operatori: Metodi di classe o esterni?

Quindi è meglio usare metodi di classe o metodi esterni per ridefinire gli operatori? Domanda aperta.

operazione : $p = p1 + p2$;

funzione somma :

```
point operator+(const point& p1, const point& p2)
```

```
{
```

```
    point ptemp(0.0,0.0);
```

```
    ptemp.x = p1.x + p2.x ;
```

```
    ptemp.y = p1.y + p2.y ;
```

```
    return ptemp ;
```

```
}
```

Le funzioni friend

Per ridefinire gli operatori fuori dalle classi (e non solo) si usano le funzioni friend.

Una funzione è detta **friend** (letteralmente “amica”) di una classe, diversa di quella eventuale di appartenenza, **se può accedere a tutti i suoi membri dichiarati private**. La funzione può essere di qualsiasi tipo, cioè una normale funzione o una funzione membro di una classe.

La dichiarazione di una funzione friend è molto semplice: basta inserire il prototipo della funzione nella definizione della classe, preceduto dalla parola chiave “friend” (non importa se nella sezione protetta o pubblica).

Le funzioni friend

```
class Point
{
    public:
    int a,b;
    void set(int i, int j)
    friend int prodotto(Point x);
};
void myclass::set(int i, int j)
{
    a = i;
    b = j;
}
int prodotto (Point x)
{
    return (x.a*x.b);
}
int main()
{
    myclass A;
    A.set(5,3);
    cout<<prodotto(A); //il risultato è 15
}
```

**Domanda aperta, si
può ridefinire un
operatore con una
funzione friend?**

Le funzioni friend (2)

```
class Point
{
    public:
    int a,b;
    void set(int i, int j)
    friend int prodotto(Point x);
};
void myclass::set(int i, int j)
{
    a = i;
    b = j;
}
int prodotto (Point x)
{
    return (x.a*x.b);
}
int main()
{
    myclass A;
    A.set(5,3);
    cout<<prodotto(A); //il risultato è 15
}
```

Domanda aperta, si può ridefinire un operatore con una funzione friend?

Sì, ma di solito si usano le funzioni friend per metodi che fanno altre operazioni sui dati!

Quali operatori si possono ridefinire?

Molti operatori possono essere ridefiniti.

- **matematici** (+ - * / %), ++, --, sia come pre incremento che come post incremento;
- **a livello del bit** (<< >> & | ^)
- **in notazione compatta** (+= -= *= /= %= <<= >>= &= |= ^=)
- **relazionali** (== != < <= > >=);
- **logici** (&& ||)
- di **serializzazione** (,)

Operator <<

Di base ridefinire l'**operator <<** significa fare una cosa equivalente alla generazione di una stringa che sarà data in output verso un file (stream) dell'istruzione `std::cout`.

```
ostream& operator<<(ostream& out, const A& a)
{
    ..... out << a.ma; (ma è un membro di A di tipo nativo)
    ..... return out ;
}
```

Se non fosse un tipo nativo A a sua volta potrebbe essere richiamato l'operatore << ridefinito.

Operator << è sempre una funzione friend, perchè restituisce un ostream.

Operatori unari ++ e -- per pre e post incremento.

Gli operatori unari non hanno la necessità di parametri di base, in quanto modificano l'oggetto stesso.

```
void operator++()  
{  
    ++this->x;  
    ++this->y;  
}
```

```
void operator++(int)  
{  
    this->x++;  
    this->y++;  
}
```

Il puntatore nascosto this

E' chiaro a tutti perchè un'**operazione** che si applica su un unico **oggetto** o che modifica il primo **operando** è preferibile che sia implementata come **metodo** della **classe**? Perchè, può sfruttare la presenza del **puntatore nascosto this**, che, come sappiamo, punta allo stesso **oggetto** della **classe** in cui il **metodo** è *incapsulato* e viene automaticamente inserito dal **C++** come primo **argomento** della **funzione**.

Ne consegue che:

1. un **operatore** in **overload** può essere implementato come **metodo** di una **classe** solo se il primo **operando** è un **oggetto** della stessa **classe**; in caso contrario deve essere una **funzione** esterna (dichiarata **friend** se accede a **membri privati**) ;
2. nella **definizione** del **metodo** se l'**operatore** è **binario**, ci deve essere un solo **argomento** (quello corrispondente al secondo **operando**), se l'**operatore** è **unario**, la **funzione** non deve avere **argomenti**.
3. se il risultato dell'operazione è l'**oggetto** stesso l'istruzione di ritorno deve essere:
return *this;

Ereditarietà in C++

Nei linguaggi di programmazione orientata agli oggetti, per ereditarietà si intende la definizione di una classe che derivi da una classe (detta classe padre).

Questo significa, creare una classe che “eredita” metodi e membri (variabili) della classe figlia, genitrice.

Ereditarietà in C++

```
class Padre
{
public:
    int a;
    void metodoPadre();
};
```

```
class Figlio : public Padre
{
public:
    int b;
    void metodoFiglio();
};
```

Ereditarietà in C++

```
class Padre  
{  
public:  
    int a;  
    void metodoPadre();  
};
```

```
class Figlio : public Padre  
{  
public:  
    int b;  
    void metodoFiglio();  
};
```

Definendo un oggetto figlio

Figlio oggetto;

è possibile accedere ad entrambi i metodi ed entrambi gli attributi presenti.

Ogni caratteristica dell'oggetto padre (definita) pubblica è anche un una caratteristica dell'oggetto figlio.

Dichiarazioni private, protected e private

Come abbiamo visto una classe può dichiarare membri e metodi pubblici, privati e protected. In caso di ereditarietà di una classe anche le dichiarazioni vengono ereditate. E' possibile inoltre effettuare una ereditarietà di tipo public, private e protected.

```
class father
{
public:
    int a;
};
```

```
class Figlio : private father
{
    int GetA() { return a; } // a si può accedere qui (a è diventato un membro privato)
}
...
{
    A a_obj;
    x = a_obj.a; // rifiutato dal compilatore: non si può accedere al membro privato a
}
```

Accessibilità delle classi

In caso di ereditarietà, la visibilità può cambiare seguendo la seguente tabella.

classe Padre	classe Figlia		
	<i>derivazione public</i>	<i>derivazione protected</i>	<i>derivazione private</i>
public	public	protected	private
protected	protected	protected	private
private	non accessibili	non accessibili	non accessibili

Regola aurea OOP C++: si mette protected tutto ciò che poi sarà ereditato dalle classi figlie. Private tutto ciò che deve rimanere incapsulato in una sola classe. Public solo l'interfaccia.

Costruttori e distruttori delle classi derivate

Definendo un oggetto figlio è possibile definire richiamare il costruttore della classe padre.

```
class Padre
{
    int a;
public:
    Padre(int aa) { a = aa; }
};

class Figlio: public Padre
{
    int b;
public:
    Figlio (int aa) : Padre (aa) {}    // Ecco un costruttore inline
    Figlio (int aa, int bb) : Padre(aa){ b = bb;}
};
```

Ereditarietà multipla

In C++ (non in Java) è possibile definire una classe figlia che eredita caratteristiche da più classi contemporaneamente.

```
class A {  
public:  
    A() { cout << "Costruttore di A" << endl; }  
};
```

```
class B {  
public:  
    B() { cout << "Costruttore di B" << endl; }  
};
```

```
class C: public B, public A {  
public:  
    C() { cout << "Costruttore di C" << endl; }  
};
```

```
int main()  
{  
    C c;  
    return 0;  
}
```

Cosa succede se definisco
C? Quali costruttori vengono
chiamati e in quale ordine?

Ereditarietà multipla, conflitti sui nomi.

In presenza di nomi uguali per metodi presenti in diverse classi ereditate è possibile risolvere manualmente eventuali conflitti sui nomi.

```
class A {  
public:  
    A() { cout << "Costruttore di A" << endl; }  
    void metodo(); { cout<<"Metodo di A" <<endl; }  
};
```

```
class B {  
public:  
    B() { cout << "Costruttore di B" << endl; }  
    void metodo(); { cout<<"Metodo di A" <<endl; }  
};
```

```
class C: public B, public A {  
public:  
    C() { cout << "Costruttore di C" << endl; }  
    void metodo(); { cout<<"Metodo di A" <<endl; }  
};
```

In pratica dico al compilatore quale metodo deve effettivamente essere chiamato.

```
int main()  
{  
    C *pc = new C;  
    pc->B::metodo();  
    pc->A::metodo();  
    return 0;  
}
```


Polimorfismo

E' una caratteristica che permette definire un oggetto di una classe derivata utilizzando un puntatore (riferimento) della classe di base.

La classe figlia può essere vista come una classe con due (o anche più!) identità. Ecco perchè questa proprietà è detta polimorfismo, parola che deriva dal greco e significa appunto "dalle molte forme".

Polimorfismo

```
class Padre
{
    ...
};
class Figlio: public Padre
{
    ...
};
...
void ExampleFunction (Padre &);
...
{
    Padre father;
    ExampleFunction (father);    // Normale chiamata
    Figlio child;
    ExampleFunction (child);    // un oggetto child è considerato come uno di tipo Padre
}
```

Child è UN (IS A) oggetto anche della classe Padre.

Polimorfismo e funzioni con lo stesso nome.

```
class Padre
{
    void metodo() { cout<<"Metodo del Padre" <<endl; }
};
class Figlio: public Padre
{
    void metodo() { cout<<"Metodo del Figlio" <<endl; }
};
...
void ExampleFunction (Padre &);
...
int main()
{
    Figlio child;
    child.metodo();
}
```

Quale metodo viene chiamato?

Polimorfismo e funzioni con lo stesso nome.

Si utilizza il metodo della classe definita nel puntatore (riferimento) della classe.

Polimorfismo e funzioni con lo stesso nome.

```
class Padre
{
    void metodo() { cout<<"Metodo del Padre" <<endl; }
};
class Figlio: public Padre
{
    void metodo() { cout<<"Metodo del Figlio" <<endl; }
};
...
void ExampleFunction (Padre &);
...
int main()
{
    Figlio child;
    Padre *punt;
    punt= &child;
    punt->metodo();
}
```

Quale metodo viene chiamato?