PROGRAMMIAMO

C++ - Overloading

Overloading di funzioni

Abbiamo già incontrato il concetto di **overloading** (letteralmente "sovraccarico") parlando dei costruttori di una classe. In sostanza è possibile definire più di un costruttore (con lo stesso nome!) per la medesima classe. Il compilatore è in grado di distinguere un costruttore dall'altro in base al numero e/o al tipo di parametri passati.

In C++ l'uso dell'overloading può essere applicato a qualsiasi funzione. Ovvero è possibile ridefinire più volte la stessa funzione con lo stesso nome, purché si usino diversi parametri. Facciamo un esempio. Sappiamo che una delle funzioni della library matematica del C++ è la *sqrt* per il calcolo della radice quadrata di un numero con la virgola con prototipo:

```
double sqrt(double);
```

Supponiamo ora di voler scrivere una funzione per il calcolo della radice quadrata intera di un numero intero, definita come il più grande intero n tale che n*n è minore o uguale al radicando:

```
int sqrt(int num)
{
int n;
for (n=0; n*n<=num;n++);
return n;
}
```

In questo modo abbiamo due funzioni con lo stesso nome *sqrt*: la funzione standard della library matematica e quella definita da noi. Ciò non comporta nessun problema, poiché il compilatore è in grado di stabilire quale funzione deve usare in base al tipo dell'argomento passato: se viene passato un valore double viene usata la funzione di library, se viene passato un int viene usata la funzione scritta da noi.

Si consideri il seguente esempio:

```
cout<<sqrt(10);    // stampa 3
cout<<sqrt(10.0); // stampa 3.16228
```

Overloading di operatori

In C++ l'overloading può essere esteso, oltre che alle funzioni, anche agli operatori. In sostanza è possibile ridefinire a piacere il funzionamento di ognuno degli operatori standard, come per esempio +, -, *, /, < , >, == eccetera.

Consideriamo per esempio di nuovo il problema di effettuare la somma di due numeri complessi, precedentemente risolto con una funzione *friend* di nome somma:

ris = somma(c1, c2);

Sarebbe tuttavia senz'altro più comodo ed elegante poter calcolare la somma fra due numeri complessi in questo modo:

ris = c1 + c2;

La sintassi necessaria per scrivere l'overloading dell'operatore + è molto simile a quella usata in precedenza per la scrittura della funzione somma, come si può notare dal confronto qui sotto:

```
complex somma(complex n1, complex n2)
   {
   complex s;
   s.a = n1.a + n2.a;
   s.b = n1.b + n2.b;
   return s;
   }
```

```
complex operator+(complex n1, complex n2)
   {
   complex s;
   s.a = n1.a + n2.a;
   s.b = n1.b + n2.b;
   return s;
   }
```

Come si vede le due definizioni sono praticamente identiche, a parte l'uso della parola chiave **operator** seguita dal simbolo dell'operatore che si vuole ridefinire.

Naturalmente affinché l'operatore + abbia diritto di accesso ai campi privati della classe *complex*, è necessario dichiararlo come friend all'interno della definizione della classe:

```
class complex {
            private: double a, b;
            public:
               complex() {a=0; b=0;}
               complex(double x, double y){a=x; b=y;}
               void set(double x, double y) {a=x; b=y;}
               void print() {cout<<a<<"+j"<<b;}
               friend complex operator+(complex, complex);
         };
```

Operatori usati come metodi di una classe

Fra i metodi di una classe possono anche essere compresi degli operatori, opportunamente ridefiniti tramite l'overloading. Consideriamo il seguente semplice esempio, mediante il quale viene ridefinito l'operatore ++ di pre-increment:

```
class complex {
            private: double a, b;
            public:
               void operator++ () {a++; b++; }
               complex() {a=0; b=0;}
               complex(double x, double y){a=x; b=y;}
               void set(double x, double y) {a=x; b=y;}
               void print() {cout<<a<<"+j"<<b;}
               friend complex operator+(complex, complex);
         };
```

A questo punto all'interno del programma è possibile usare l'operatore nel seguente modo:

```
int main(int argc, char *argv[])
{
complex num(1,3);
++num;
...
```

Il risultato dell'esecuzione è ovviamente quello di incrementare di uno sia la parte reale che la parte immaginaria del numero complesso *num*.

Il codice precedente tuttavia non funziona in un caso un po' più complicato come il seguente:

```
int main(int argc, char *argv[])
{
complex num(1,3), num2;
num2 = ++num;
...
```

In questo caso l'assegnazione a num2 produce un errore, dovuto al fatto che l'operatore ++ è stato dichiarato come void (cioè che non torna nessun valore). Per far funzionare il programma bisogna modificare la dichiarazione dell'operatore nel seguente modo:

```
class complex {
            private: double a, b;
            public:
               vcomplex operator++ () {a++; b++; return *this;}
               complex() {a=0; b=0;}
               complex(double x, double y){a=x; b=y;}
               void set(double x, double y) {a=x; b=y;}
               void print() {cout<<a<<"+j"<<b;}
               friend complex operator+(complex, complex);
        };
```

La parola riservata **this** del linguaggio C++ indica l'indirizzo della classe di appartenenza (dunque this è un puntatore implicito all'oggetto corrente). L'asterisco davanti (**this*) serve per trasformare il puntatore in un'istanza della classe.

Si osservi di passaggio che volendo effettuare l'overloading del postincrement (cioè num++ invece di ++num), la dichiarazione dell'operatore dev'essere la seguente:

```
complex operator++(int) {a++; b++; return *this;}
```

Si noti la dichiarazione int fra parentesi tonde che serve appunto per distinguere il post dal preincrement.

precedente - successiva