SOVRACCARICAMENTO DEGLI OPERATORI

Roberto Nardone, Luigi Romano

ROADMAP

- Concetti di base
- Funzioni e Classi Friend
- Introduzione al Sovraccaricamento (overloading) degli Operatori
- Quando utilizzare l'overloading, esempio dei numeri complessi
- Sovraccaricamento di alcuni operatori unari e binari
- Casi particolari:
  - Overload dell'operatore di assegnamento
  - Overload degli operatori di stream
- Sovraccaricamento attraverso funzioni friend

LE FUNZIONI FRIEND

- Nella definzione di una classe è possibile elencare quali funzioni, **esterne alla classe**, possono accedere ai membri privati della classe. Queste funzioni sono dette **friend** (amiche).
- Per dichiarare una funzione friend è necessario includere il prototipo nella classe, facendola precedere dalla parola chiave **friend:**

```
class MyClass {
      …
      friend Tipo funz(…);
      …
};
```

- Le funzioni friend sono particolarmente utili quando due o più classi contengono membri correlati con altre parti del programma.

ESEMPIO FUNZIONI FRIEND


```
class Myclass {
  int a,b;
public:
  void set_ab(int i, int j)
  friend int sum(Myclass x);
};
```

```
main()
{
  Myclass n;

 n.set_ab(2, 4);

cout<<sum(n);
}
```

```
Myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// sum non è membro della classe
int sum(Myclass x)
{
 return x.a + x.b;
}
```

La funzione **Sum** non è membro della classe, ma essendo friend può accedere ai suoi membri privati

CLASSI FRIEND

- In C++ è anche possibile rendere un'intera classe **friend** di un'altra classe.
- In tal caso tutte le funzioni della classe dichiarata **friend** avranno accesso ai membri privati della classe.
- La dichiarazione di classe friend è del tipo:

```
class MyClass {
      …
      friend class C2;
      …
};
```

- Osserviamo che le funzioni membro di C2 possono accedere ai membri di C1 e non viceversa.

INTRODUZIONE ALL'OVERLOADING

- Il meccanismo di **sovraccaricamento** (o **overloading) degli operatori** consente di attribuire ulteriori significati agli operatori del linguaggio (ad es. + / - *)
- In C++ è possibile eseguire l'overloading di molti operatori (non tutti) per consentire loro di svolgere operazioni specifiche rispetto a determinate classi
- L'overlading di un operatore, estende l'insieme dei tipi al quale esso può essere applicato, lasciando invariato il suo uso originale.
- L'overloading degli operatori è alla base delle operazioni di I/O del C++

OVERLOADING OPERATORI IN C++

- L'overloading degli operatori viene realizzato per mezzo delle funzioni ***operator***.

- Un funzione *operator* definisce le specifiche operazioni che dovranno essere svolte dall'operatore sovraccaricato (overloaded) rispetto alla classe specificata.

- Ci sono due modi per sovraccaricare un operatore.
  - Tramite **funzioni membro della classe**;
  - Tramite **funzioni esterne** che però devono essere definite **friend** per la classe.

SINTASSI DI SOVRACCARICAMENTO


```
Tipo nome-classe::operator#(Tipo1 arg1, Tipo2 arg2, ...) {
      // istruzioni
            …
}
dove # è il simbolo dell'operatore da sovraccaricare
```

- Nella maggior parte dei casi le funzioni operator restituiscono un oggetto della classe su cui operano, ma in generale possono restituire qualsiasi tipo valido.
- Quando si esegue l'overloading di un **operatore binario**, la funzione operator ha un solo argomento, mentre se si tratta di un **operatore unario** la funzione non ha argomenti.

ALCUNE REGOLE

- È possibile modificare il significato di un operatore esistente
- Non è possibile creare nuovi operatori e non è opportuno ridefinire la semantica di un operatore applicato a tipi predefiniti
- Non è possibile cambiare precedenza, associatività e numero di operandi
- Non è possibile usare argomenti di default.

DIETRO LE QUINTE

- Ma effettivamente cosa succede quando si effettua l'overloading di un operatore?
- L'overloading degli operatori è realizzato dal **compilatore**
- Supponiamo di avere un'espressione della forma:

oggetto1 + oggetto2

- Senza una ridefinizione dell'operatore, il compilatore restituirebbe errore poichè l'espressione di somma tra i due oggetti non sa come intepretarla
- Il compilatore prima di lanciare l'errore verifica se:
  - Nella classe **classe1** dell'oggetto **oggetto1** vi è una funzione membro della forma: *operator op(classe2)*
  - se vi è una funzione non membro (**friend**) della forma: *operator op(classe1, classe2)*

ESEMPIO DI UTILIZZO

- L'esempio più comune di utilizzo dell'overloading degli operatori è quello dei **numeri complessi**.
- Un numero complesso infatti può essere rappresentato attraverso una classe contenente due variabili membro, una relativa alla **parte reale** ed una relativa alla **parte immaginaria**.
- Il linguaggio non ha una definizione di default per gli operatori di questo tipo

OVERLOADING OPERATORE DI SOMMA – 1/2


```
class Complex {
    float re;
    float im;
public:
    Complex(float r = 0.0, float i = 0.0) {
        re = r;
        im = i;
    }

    float getRe() const {
        return re;
    }
    float getIm() const {
        return im;
    }
    void setRe(float r) {
        re = r;
    }
    void setIm(float i) {
        im = i;
    }
    void show();
    Complex operator+(Complex op2);
};
```

```
void Complex::show() {
    cout << endl << "re: " << re << " im: " << im;
}

Complex Complex::operator+(Complex op2) {
    Complex tmp;
    tmp.re = re + op2.re;
    tmp.im = im + op2.im;
    return tmp;
}
```

OVERLOADING OPERATORE DI SOMMA – 2/2

- Da notare che l'esecuzione della linea di codice:

```
c1=c2+c3;
```

- Per il compilatore sarà come:

```
c1 = c2.operator+(c3);
```

```
int main(){

    Complex c1, c2(1, 1), c3(4,5);

    c1.show();
    c2.show();
    c3.show();

    c1 = c2 + c3;
    c1.show();

    return 0;
}
```

ALCUNE OSSERVAZIONI

- Quando si esegue l'overloading di un operatore binario è **l'oggetto di sinistra** a generare la chiamata alla funzione operator.
- L'operatore di **assegnamento** = può essere usato solo perchè operator+ restituisce un oggetto della classe Complex.
- La funzione operator+ NON modifica gli operandi. In generale, è opportuno definire sempre delle funzioni operator che non modificano gli operandi, in analogia con gli operatori standard.

RIDEFINIZIONE DI ALTRI OPERATORI


```
Complex Complex::operator-(Complex op2) {
    Complex tmp;
    tmp.re = re - op2.re;
    tmp.im = im - op2.im;
    return tmp;
}

Complex Complex::operator++() {
    re++;
    im++;
    return *this;
}
```

- Poiché è l'oggetto di sinistra a generare la chiamata a operator- i dati di op2 devono essere sottratti a quelli dell'oggetto chiamante, al fine di conservare la semantica della sottrazione
- Per l'operatore di incremento non ci sono parametri poiché è un operatore unario.
  In questo caso viene modificato l'operando.

Operatore di Assegnamento – 1/2

- E’ un operatore binario, che, nel suo significato naturale, copia il secondo operando nella locazione di memoria rappresentata dal primo.
- **In assenza di overload**, l’**operatore di assegnazione funziona in C++ anche per le classi**, nel senso che esegue una copia degli oggetti membro a membro. **Se alcuni membri sono puntatori,** la semplice copia genera **due problemi**:
  - Dopo la copia, l'area precedentemente puntata dal primo operando resta ancora, cioè occupa spazio, ma non è più accessibile;
  - Il fatto che due oggetti puntino alla stessa area è pericoloso, perché, se viene chiamato il distruttore per uno dei due, l'altro si ritrova a puntare a un'area della heap che non è più disponibile.

Operatore di Assegnamento – 2/2

- E’ necessario in questi casi che l’operatore di assegnazione non esegua la copia del puntatore…
- …ma dell’aria puntata, che deve essere allocata separatamente dalla prima con l’operatore **new**
  (e quindi il contenuto dei due puntatori risulterà diverso).
- L'area precedentemente puntata dal primo operando deve essere deallocata con l'operatore delete.

ESEMPIO OPERATORE DI ASSEGNAMENTO

- **Ad esempio**: Per l'esecuzione di istruzioni del tipo: **a2=a1;**
- Dove **a1** e **a2** sono istanze della classe **A** costituita da un solo membro **pa** (un puntatore a int).
- Il corretto overload dell'operatore si fa come segue:

```
A& A::operator=(const A& a) {
    if (this == &a)
        return *this;
    delete[] pa;
    pa = new int;
    *pa = *a.pa;
    return *this;
}
```

Esempio operatore di Assegnamento per il caso dei Numeri Complessi

- Nel nostro esempio dei numeri complessi non c'era nessun puntatore come variabile membro. Dunque non sarà necessario quanto detto prima.
- Basterà semplicemente copiare le due variabili membro.
- L'operatore restituisce ***this** ovvero l'oggetto che ha generato la chiamata. Questo accorgimento rende possibile assegnamenti multipli del tipo:

c1 = c2 = c3;

```
Complex Complex::operator=(Complex op2) {
    re = op2.re;
    im = op2.im;
    return *this;
}
```

OVERLOAD OPERATORI DI STREAM

- Un caso particolare è rivestito dagli operatori di flusso (<< >>)
- Un'operazione di **output** viene eseguita tramite l'operatore <<, che "inserisce" nell'oggetto cout (primo operando) il dato da scrivere (secondo operando), il quale può essere di qualunque tipo nativo (sono riconosciute anche le stringhe)
- E' **possibile estendere l'operazione anche ai tipi astratti**, per esempio per far sì che l'operazione:
  - cout<<a; (dove a è un'istanza di una classe **A**)
- Generi su video una stampa dei valori assunti dai **membri** di **a**.

ESEMPIO OVERLOAD DI <<

- cout, oggetto globale generato all'inizio dell'esecuzione del programma, é un'istanza della classe **ostream**, che viene detta "**classe di flusso di output**" (e dichiarata in <iostream.h>)
- Il primo argomento della funzione dovrà essere lo stesso oggetto cout, mentre il secondo argomento dovrà essere l'**oggetto a** da trasferire in output.
- Come per l'operatore di assegnamento, la funzione dovrà restituire lo stesso **cout**, per permettere l'associazione di ulteriori operazioni nella stessa istruzione.

Esempio degli Operatori di Stream per il caso dei Numeri Complessi


```
ostream& operator<<(ostream& os, Complex op) {
    os << op.getRe();
    if (op.getIm() > 0)
        os << " + ";
    else if (op.getIm() < 0)
        os << " - ";
    os << op.getIm();
    return os;
}

istream & operator>>(istream & in, Complex & op) {
    Complex tmp;

    float re, im;
    in >> re;
    tmp.setRe(re);

    in >> im;
    tmp.setIm(im);

    op = tmp;
    return in;
}
```

- Notare che:
  - il primo argomento della funzione appartiene a ostream/istream e non ad A e quindi deve essere definita come funzione esterna **friend** di A
  - Il C++ non ammette la creazione di copie dell'oggetto cout

```
friend ostream& operator<<(ostream& os, Complex op);
friend istream& operator>>(istream & in, Complex & op);
```

Overloading attraverso funzioni friend

- Come detto in precedenza l'overloading può essere eseguito anche per mezzo di funzioni non membro, ma definite **friend** per la classe in esame.
- Nel caso delle funzioni **friend** il numero di argomenti coincide con il numero di operandi.
- Dunque se volessimo realizzare la somma tra due complessi la funzione di overload avrà due argomenti, uno per ogni operando complesso
- Le funzioni friend offrono in generale **maggiore flessibilita.**
- Consideriamo il caso in cui alla classe Complex sia stata definita un'ulteriore funzione membro operator+ che prende come operando un **float**

VANTAGGIO NELL'USO DELLE FUNZIONI FRIEND – 1/2


```
class Complex {
      .
      .
      .
  Complex operator +(float val);
};
```

```
Complex operator+(float val)
{
  Complex tmp;

  tmp.re = re +val;
  tmp.im = im +val;

  return tmp;
}
```

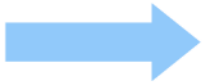
```
Main()
{
  Complex c1, c2;

  c2 = c1  + 100; // OK
  c1 = 100 + c1;  // ERRORE!
}
```

VANTAGGIO NELL'USO DELLE FUNZIONI FRIEND – 2/2

- E' possibile rimediare al problema precedente proprio attraverso le funzioni **friend**

```
class Complex {
      .
      .
      .
  friend Complex operator +(Complex op, float val);
  friend Complex operator +(float val, Complex op);
};
```

```
Complex operator+(Complex op, float val)
{
  Complex tmp;

  tmp.re = op.re +val;
  tmp.im = op.im +val;

  return tmp;
}

Complex operator+(float val, Complex op)
{
  Complex tmp;

  tmp.re = op.re +val;
  tmp.im = op.im +val;

  return tmp;
}
```



```
Main()
{
  Complex c1, c2;

  c2 = c1   + 100; // OK
  c1 = 100 + c1;   // OK
}
```

ESERCIZIO

- Riscrivere l'esempio dei numeri complessi facendo uso però delle funzioni friend