



UNIVERSITÀ
degli STUDI
di CATANIA

Overloading degli operatori

Corso di programmazione I AA 2019/20

Corso di Laurea Triennale in Informatica

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: gfarinella@dmf.unict.it

Dipartimento di Matematica e Informatica

1. Introduzione
2. Regole base per overloading operatori
3. Implementazione mediante funzioni non membro e friend
4. Implementazione mediante funzioni membro
5. Operatori Speciali
6. Operatori cast e IO

Introduzione

Operatori overloaded

Un operatore *overloaded* è un operatore che **assume differenti significati** in base al **tipo di operandi**.

Tra gli operandi, **almeno uno** è di un tipo definito dall'utente (oggetto).

Esempio: overloading operatore **+** per concatenazione stringhe.

```
1  string x = "Hello";  
2  string y = "World";  
3  
4  string z = x + y;
```

Vantaggi operatori overloaded

1. **Notazione concisa.**
2. **Notazione fortemente espressiva.**

Esempio: somma tra matrici.

```
1 Matrix A, B;  
2 //...  
3 Matrix Z = A + B;
```

VS

```
1 Matrix A, B;  
2 //...  
3 Matrix Z = A.sum(B);
```

Overloading operatore andrebbe definito quando **la semantica dell'operazione** su oggetti è facilmente **riconducibile da semantica originale** dell'operatore stesso.

Esempio: *operatore* "+" \Rightarrow concatenazione stringhe.

Regole base per overloading operatori

Regole per overloading operatori

Operatori per i quali è possibile definire overloading:

Tabella operatori.

+	-	*	/	%	^
	~	!	=	<	>
-=	*=	/=	%=	^=	&=
<<	>>	>>=	<<=	==	!=
<u>>=</u>	&&		++	--	->
->	[]	()	new	new []	delete
&	+=	=	<=	,	delete []

Regole per overloading operatori

Operatori per i quali NON è possibile ridefinire overloading:

- Operatore risoluzione di scope ::
- Operatore di selezione membro .
- Operatore di selezione membro mediante puntatore .*
- Operatore ternario ? :
- **sizeof**
- **typeid**

Regole per overloading operatori

Inoltre:

- **Non è possibile** definire nuovi operandi.
- **Non si può modificare il numero di operandi** di un operatore:
 - ES: Overloading operatore $+$ deve prevedere due operandi!
- **Non si può modificare precedenza tra operatori.**
 - ES: $a + b * c$ regola di precedenza operatore $*$ continua a valere. rispetto a $+$
- **Non si può modificare associatività tra operatori.**
 - ES: $a + b + c$ equivalente a espressione $(a + b) + c$.

Regole per overloading operatori

Un operatore overloaded può essere implementato come

- funzione membro non-static;
- funzione non membro che ha almeno un parametro formale del tipo definito dall'utente;

Sintassi prototipo:

<type> **operator** <op> <(...)>

operator è una keyword del C++.

op è il nome dell'operatore (vedi Tabella operatori).

type il tipo di ritorno.

Implementazione mediante funzioni non membro e friend

Classe Vettore3D.


```
classe Vettore3D{  
    float x, y, z;  
    //..  
}
```

Si immagini di voler esprimere operazione di somma di matrici mediante overloading operatore binario +.

Overloading: funzioni non membro

ES: Operatore binario +.

```
Vettore3D operator+ (const Vettore3D sx,  
                     const Vettore3D dx);
```



- Funzione **non membro**.
- I due **argomenti** sono i due **operandi** operatore **binario** somma, istanze di Vettore3D
- Se dichiarata friend della classe Vettore3D, allora accesso garantito a tutti i membri (pubblici e privati) di Vettore3D.

Overloading: funzioni non membro

Quindi: non-membro

```
Vettore3D operator+ (const Vettore3D sx ,  
                    const Vettore3D dx);
```

friend:

```
class Vettore3D{  
    //...  
→ friend Vettore3D operator+ (const Vettore3D sx ,  
                             const Vettore3D dx);  
  
    //...  
}
```

Overloading: funzioni non membro

Invocazione operatore binario, non membro

```
1  Vettore3D a(1,2,3);  
2  Vettore3D b(3.4, 2.3, 5);  
3  
4  Vettore3D c = operator+(a,b);  ↑ ↑  
5  // oppure  
6  Vettore3D c = a + b;  ↑
```


Overloading: funzioni non membro

Altro esempio: operatore **unario** `-`.

```
Vettore3D operator- (const Vettore3D obj);
```


- Funzione **non membro**.
- Unico argomento di tipo `Vettore3D`.
- Se dichiarato friend di `Vettore3D`, allora accesso garantito a tutti i membri (pubblici e privati) di `Vettore3D`.

Invocazione operatore unario “-”, non membro

```
1  Vettore3D a(1,2,3);  
2  
3  Vettore3D b = operator-(a)  
4  // oppure  
5  Vettore3D b = -a;
```

Risoluzione overloading con funzioni non-membro

Si supponga di avere un **costruttore con argomenti standard** e **overloading** operatore come funzione **non-membro**.

```
1  class Vettore3D{  
2   Vettore3D(float a=0.0, float b=0.0, float c=0.0);  
3  }  
4  //..  
5  Vettore3D operator+(Vettore3D a, Vettore3D b);  
6  //..
```

Risoluzione overloading con funzioni non-membro

Espressioni alle righe 3 e 4 sono lecite per il compilatore, che applica regole **risoluzione funzioni overloaded**

```
1  Vettore3D a = Vettore3D (1.4 ,2 ,3.2 );  
2  //..  
3  Vettore3D b = 11.8 + a; // 11.8 → Vettore3D  
4  Vettore3D c = a + 7.9; // 7.9 → Vettore3D
```

In entrambi i casi, il compilatore esegue **conversione float → Vettore3D** del parametro attuale per ottenere oggetto Vettore3D.

- Numero di argomenti corrispondente al numero di operandi.
- Se dichiarato **friend** di una classe, **accesso a tutti i membri della classe stessa**.
- Se costruttore classe lo permette, il compilatore applica **conversioni** su uno degli argomenti.

Sintassi invocazione

Operatore	Invocazione	
binario “@”	operator@(a,b)	a @ b
unario “@”	operator@(a)	@a

Esempi svolti

28_01.cpp – Overloading, **funzioni non-membro**.

28_02.cpp – Overloading, **funzioni friend**.

Implementazione mediante funzioni membro

Overloading: funzioni membro

```
1  class Vettore3D{  
2      // ...  
3  → Vettore3D operator + (const Vettore3D); // binario  
4      Vettore3D operator - ( ); // unario  
5  }
```

- L'operando sinistro è implicitamente collegato al puntatore this. Di conseguenza:
 - **Operatore binario** prevede un solo **parametro formale**;
 - **Operatore unario** prevede **zero parametri formali**;

Invocazione operatore binario

```
1  Vettore3D a(1,2,3);  
2  Vettore3D b(3.4, 2.3, 5);  
3  
4  Vettore3D c = a.operator+(b)  
5  // oppure  
6  Vettore3D c = a + b;
```

Invocazione operatore unario

```
1  Vettore3D a(1,2,3);  
2  
3  Vettore3D b = a.operator-()  
4  // oppure  
5  Vettore3D b = -a;
```

Risoluzione overloading con funzioni membro

Si supponga di avere un costruttore con argomenti standard e **overloading operatore implementato come funzione membro**.

```
1  class Vettore3D{
2      Vettore3D(float a=0.0, float b=0.0, \
3          float c=0.0);
4      //...
5      Vettore3D operator+(Vettore3D operando);
6  }
```

Risoluzione overloading con funzioni membro

Espressione alla riga 3 lecita: il compilatore applica regole **risoluzione funzioni overloaded**:


- conversione 7.9 → Vettore3D
- NB: 7.9 è operando destro

```
1  Vettore3D a = Vettore3D (1.4 , 2 , 3.2);  
2  //..  
3  Vettore3D b = a + 7.9; //OK  
4  //equivalente a:  
5  Vettore3D b = a.operator+(7.9); //OK
```

Risoluzione overloading con funzioni membro

Espressione alla riga 3 genera ERRORE di compilazione:

```
1  Vettore3D a = Vettore3D (1.4 ,2 ,3.2 );  
2  //..  
3  Vettore3D b = 7.9 + a; //Comp. ERR!!
```





Il compilatore ha a disposizione una funzione di overloading (membro) in cui operando di sinistra è **implicitamente** il puntatore `this`.

Nessuna conversione di operando a sinistra!!

Overloading: funzioni membro. Remarks

- **Primo argomento implicito**, esso corrisponde al puntatore `this` dell'oggetto sul quale viene invocato.
- **Se costruttore classe lo permette**, il compilatore applica **conversioni**:
 - conversione applicata solo su argomenti che abbiano un **corrispondente parametro esplicito**.

Sintassi invocazione

Operatore		Invocazione	
 binario "@"		<code>a.operator@(b)</code>	<code>a @ b</code>
 unario "@"		<code>a.operator@()</code>	<code>@a</code>

Esempi svolti

28_03.cpp – Overloading, **funzioni membro**.

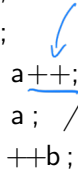
Operatori Speciali

Operatori speciali “--” e “++”

Il compilatore non interpreta allo stesso modo operatore di **incremento/decremento unario** se in forma **postfissa** o **prefissa**.

Esempio:

```
1  int a=0;
2  int b=0;
3  cout << a++; //stampa 0
4  cout << a; // stampa 1
5  cout << ++b; // stampa 1
```



Operatori speciali “--” e “++”

Di conseguenza, compilatore impone al programmatore di “discriminare” forma prefissa e postfissa nella definizione di overloading di tali operatori.

```
1 → Vettore3D &operator--(); // prefisso  
2   Vettore3D operator--(int); // postfisso  
3 → Vettore3D &operator++(); // prefisso  
4   Vettore3D operator++(int); // postfisso
```

Operatori speciali “--” e “++”

```
1  Vettore3D &operator--(); // prefisso
2  Vettore3D operator--(int); // postfisso
3  Vettore3D &operator++(); // prefisso
4  Vettore3D operator++(int); // postfisso
```

Operatore postfisso prende parametro formale “dummy” (int) per discernere da operatore prefisso.

NB: Nessun parametro attuale corrisponderà tale parametro formale.

Operatori speciali “--” e “++”

Esempi svolti

28_04.cpp – Overloading, **operatore incremento postfisso e prefisso.**

Overloading operatore di assegnamento “=”

Overloading operatore “=” si rende necessario quando inizializzazione membro a membro da altro oggetto non sufficiente:

- ad esempio la classe contiene puntatori ad aree di memoria allocate dinamicamente.

```
1  class X{
2      X(int , float );
3      X (const X &);
4  }
5  X x1{1, 4.5};
6  X x2 {x1}; //costruttore di copia
7  X x3 = x1; //costruttore di copia
8  x3 = x2; //inizializzazione memberwise!
```

Overloading operatore di assegnamento “=”

Se definito anche overloading operatore di assegnamento “=”:

```
1  class X{
2      X(int , float );
3      X (const X &); // costr. copia
4      X &operator=(const X&) const;
5  }
6  X x1 {1, 4.5};
7  X x2 {x1}; //costruttore di copia
8  X x3 = x1; //costruttore di copia
9  x3 = x2; //x3.operator=(x2);
```

allora la copia avverrà nel modo “corretto” anche a seguito di riassegnamenti, successivi a creazione dell’oggetto.

Overloading operatore di assegnamento “=”

Esempi svolti

28_05.cpp – Overloading operatore “=”

Come gli operatori “++” (incremento) e “--” (decremento), esistono altri operatori definiti **speciali**:

- operatore “freccia” (*dereferencing operator*) “- >”.
- operatore “[]” (*subscript* o indicizzazione) e “()” (*function call*);
- operatori new e delete

Overloading Operatore “– >”:

- va implementato come **funzione membro**;
- Operatore **unario postfisso**;
- Tipo di ritorno: puntatore a oggetto oppure oggetto al quale si può applicare operatore “– >”;

Overloading Operatore “– >”:

- Se definito in una classe X:
 - utile per accesso diretto a membri di oggetti referenziati da X;
 - “smart pointer” per accesso veloce a interfaccia di oggetti referenziati in X;
 - di conseguenza, interfaccia di tali oggetti accessibile da istanza di X;

Operatori speciali [], (), new, delete, - >

```
1  class Y{
2      //...
3      public: void foo();
4  }
5  class X{
6      Y* y; ←
7      //...
8      Y* operator->(){ // overloading operatore "->"
9          return y;
10     }
11 }
12 //...
13 X x;
14 x -> foo(); // (x.operator->())->foo()
```

Esempi svolti

28_06.cpp – Overloading, **operatore freccia** (*dereferencing*);

Operatori speciali [], (), new, delete, – >

Operatore di indicizzazione (*subscript*):

- va implementato come **funzione membro**;
- deve prevedere **un parametro formale**, di qualunque tipo;
- se tipo di ritorno è reference, allora potrà essere usato nella parte sinistra di un'operazione di assegnamento (*lvalue*).

```
1  class Array{  
2      float  vettore [DIM];  
3      //..  
4      float  &operator [] (const int );  
5  }
```

Esempi svolti

28_07.cpp – Overloading, **operatore indicizzazione**;

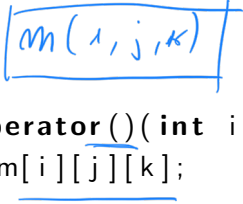
Overloading Operatore “()” (*function call* o *application operator*):

- Permette di usare la notazione di **chiamata a funzione**: expression(expression_list), dove expression primo argomento, expression_list è il secondo argomento;
- Spesso Usato per **indicizzare dati** di un oggetto mediante più indici;
- Usato anche per **operare su un oggetto come se fosse una funzione**;

Operatori speciali [], (), new, delete, - >

Overloading Operatore “()” (*function call* o *application operator*) (cont.)

```
1 class Matrix3D{  
2     float ***m;  
3     //...  
4     public:  
5         float &operator()(int i, int j, int k) {  
6             return m[i][j][k];  
7         }  
8 }
```



Esempi svolti

28_08_esempio_completo.cpp – Overloading di diversi operatori, tra cui operatore *function call*;

Operatori speciali [], (), new, delete, – >

Overloading operatori new e delete.

Nella forma più semplice, i prototipi sono nella seg. forma.

```
1  void operator delete ( void* ptr );  
2  void operator delete [] ( void* ptr );  
3  void* operator new ( std::size_t count );  
4  void* operator new [] ( std::size_t count );
```

Documentazione operatori gestione memoria (header <new>):

<https://en.cppreference.com/w/cpp/header/new> 

Per usare implementazione standard di new e delete, non necessario includere header <new>.

Overloading operatori new e delete (cont.).

Programmatore potrebbe:

- **ridefinire** operatori new e delete **globalmente** (poco raccomandato)
- ridefinire operatori new e delete **localmente** ad uno scope (e.g. per una specifica classe)

Overloading operatori new e delete (cont.).

Ridefinire new e delete **localmente**:

- **Implicitamente** membri **static**;
- new deve allocare memoria e restituire un puntatore “generico” (void);
- delete deve deallocare memoria a partire da un certo puntatore;

Esempi svolti

28_09.cpp – Overloading (banale) di operatori new e delete

Operatori cast e IO

Overloading operatore di cast

Conversioni di tipo implicite:

- Conversioni tra **tipi primitivi** del C (1)
- Chiamata a **costruttore** con un parametro (2)
- **Overloading** operatore di **cast** (3)

Casi (1) e (2)

```
1  int a = 2.0 + 3; // (1): 2.0 (double) —> 2 (int)
2
3  class ClassX{
4      ClassX(int a=0, int b=1); //argomenti standard
5  }
6  ClassX c = b + 1; // (2): 1 (int) —> ClassX
```

Overloading operatore di cast

Caso (3) (Overloading operatore di cast). Tipi di conversione:


- a. operare conversione oggetto → tipo primitivo;
- b. operare conversione (user-defined) X → (user-defined) Y;
- a (oggetto → tipo primitivo)

```
1  class X{  
2      int i;  
3      //...  
4      public:  
5          operator int () const { return i; }  
6  }
```


Overloading operatore di cast

b. conversione (user-defined) **X** \rightarrow (user-defined) **Y**;

```
1  class Y{  
2      float j;  
3      //...  
4      public:  
5          Y(float x);  
6  }  
7  class X{  
8      int i;  
9      //...  
10     public:  
11         operator Y() const{ return Y((float) i + 0.5); }  
12     }
```



Esempi svolti

28_10.cpp – Conversioni implicite vs conversioni esplicite

28_11.cpp – Overloading operatore cast, conversione implicita

28_12.cpp – Overloading operatore cast, conversione esplicita

Overloading operatori << e >>

Operatori “<<” (inserimento) e “>>” (estrazione) spesso usati con oggetti `cout` e `cin`, `stringstream`, etc.

```
1  cout << " Hello World" << endl ;  
2  cin >> s ;
```

`cin` è istanza (statica, ovvero **scope globale**) di classe `istream`;

`cout` è istanza (statica, ovvero **scope globale**) di classe `ostream`;

Overloading operatori << e >>

Classi `istream` e `ostream` costituiscono le **classi di base** per IO su libreria standard C++;

Classi `istream` e `ostream` incapsulano codice relativo a gestione dei **flussi** (o canali) di IO;

Flussi rappresentano modalità di IO a caratteri (byte), e sono **indipendenti dai dispositivi** usati di volta in volta (video, tastiera, schede di rete, etc);

Oggetto **`cin`** opera sullo **standard input**, **`cout`** opera sullo **standard output**;

Standard input generalmente associato a tastiera, standard output generalmente associato a video.

Overloading operatori << e >>

Overloading operatori inserimento ed estrazione

```
1 ostream & operator<< (ostream &, const X&);  
2 istream & operator>> (istream &, X&);
```

Punti chiave:

- entrambe le forme restituiscono reference allo stream ricevuto come input;
- per lettura dati da oggetto ed **inserimento nello stream di output** (operatore "<<"), secondo parametro **const**;
- per estrazione dati dallo stream di input ed "inserimento" nello operando oggetto (operatore ">>"), secondo parametro **non-const** (oggetto va modificato!);

Overloading operatori << e >>

Overloading "<<".

```
1  class X{
2      int i;
3      float j;
4      //...
5      friend ostream& operator<<(ostream &s, X& x);
6  }
7  ostream& operator<<(ostream &s, X& x){
8      s << x.i << " ," << x.j;
9      return s;
10 }
11 //..
12 X a,b,c;
13 cout << a << " ," << b << " ," << c << endl;
```

Overloading operatori << e >>

```
1  friend ostream& operator<<(ostream &s, X& x);
2
3  ostream& operator<<(ostream &s, X& x){
4      s << x.i << " , " << x.j;
5      return s;
6  }
7
8  cout << a << " , " << b << " , " << c << endl;
```

Domanda: perchè friend? Overloading prende un parametro `ostream` oppure `istream` a sinistra. MA una funzione membro avrebbe operando di sinistra implicitamente associato a puntatore `this`!

Overloading operatori << e >>

```
1  friend ostream& operator<<(ostream &s, X& x);
2
3  ostream& operator<<(ostream &s, X& x){
4      s << x.i << " , " << x.j;
5      return s;
6  }
7
8  //cout << a << " , " << b << " , " << c << endl;
9  //equivalente a...
10 ((((((cout << a) << " , ") << b) << " , ") << c) << endl;
```

Operatori "<<" e ">>" **associativi** a sinistra.

Esempi svolti

28_13.cpp – Overloading operatori << e >>.

Homework H28.1

Con riferimento alla classe `Matrice3D` (homework 24.1), implementare le seguenti ulteriori funzionalità:

- Costruttore di copia;
- Overloading operatore di assegnamento "=", per gestire correttamente la copia degli elementi della matrice sorgente (operando destro) nella matrice destinazione (operando sinistro);
- Overloading operatore di uguaglianza "==", che restituisca true solo se le due matrici hanno identiche dimensioni e identici valori;

Homework H28.1

- Overloading operatore di disuguaglianza “!=”, duale rispetto all'operatore “==”;
- Overloading operatore “()”, che permetta di indicizzare i singoli valori della matrice mediante tre indici; inoltre, tipo di ritorno deve essere reference all'elemento estratto, in modo che valore di ritorno si possa usare nella parte sinistra di una espressione di assegnamento;
- Overloading operatore “+”, che permetta di sommare due matrici con la stessa dimensione;
- Overloading operatore “*”, che permetta di eseguire il prodotto “riga per colonna” di due matrici;

Homework H28.1

- Overloading operatore “<<”, che permetta di “stampare” gli elementi della matrice (suo operando destro) su uno stream di output (suo operando sinistro);

Note.

- Usare il passaggio per riferimento dei parametri formali, quando possibile e/o opportuno;
- Usare il modificatore `const` per i parametri formali, quando opportuno (quando gli operandi non andrebbero modificati dalla funzione di overloading);

Homework H28.1

- Implementare ogni eventuale metodo che si renda necessario per la corretta realizzazione delle funzionalità e/o per garantire la necessaria modularità (ES: metodo relativo alla gestione della memoria dinamica che possa essere invocato sia da costruttore che da costruttore di copia);

FINE