

UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Overloading degli operatori

Stefano Ghidoni



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE



# Agenda

- Overloading degli operatori
  - Helper function
  - Funzioni membro
- Esempi



# Reimplementare gli operatori

- Recall: espressioni e operatori
- Gli operatori sono elementi fondamentali nelle espressioni
- Operatori predefiniti per i tipi built-in
- Che succede agli UDT?
  - È possibile usare gli operatori?
  - Cosa significa sommare due date?
- Possibile se ne è definito il comportamento



# Overloading degli operatori

- In C++ è possibile implementare quasi tutti gli operatori per operandi di tipo definito dall'utente (class o enum) *Gli operatori pre-esistenti, non nuovi, visto che renderebbe confuso il codice*
- **Operator overloading**
- Può essere effettuato usando una funzione
  - Helper function
  - Funzioni membro
- Overloading solo di operatori esistenti
  - Con lo stesso numero di operandi e la stessa sintassi



# Overloading degli operatori

- L'overloading è implementato creando una funzione con un nome specifico
  - operator+, operator++, operator\*, operator[], ...
- Il C++ riconosce questo pattern e traduce

```
Date d {2010, Mon::feb, 21};  
  
++d; // equivalente a operator++(d),  
// oppure a d.operator++()
```

Il primo caso è una funzione che accetta un oggetto di tipo date, nel secondo caso è una funzione membro.



# Overloading degli operatori

- Deve essere presente almeno un argomento UDT
  - Non è possibile definire `int operator+(int, int);`
- Uno strumento potente, ma:
  - È importante non definire operatori con significati contro-intuitivi *è buona pratica di programmazione*
  - È sensato definire gli operatori solo se hanno concettualmente senso per quel tipo di dato
- Gli operatori più utili: `=`, `==`, `!=`, `<`, `[]`, `()`



# Overloading degli operatori

- Ogni operatore ha il suo pattern
  - Il pattern è diverso se è funzione membro o funzione esterna

```
int operator+(int, int);      // errore: + built-in  
  
vector operator+(const Vector&, const Vector&);    // OK  
  
vector operator+=(const Vector&, int);                // OK
```

→ Non è uno standard vector, è un oggetto molto simile che definiremo noi in seguito.  
Non sarebbe possibile definirla tra gli standard vector.



# Overloading: helper function

- Pattern di operatori implementati con helper function
  - operator+, operator-, operator\*, operator/: due argomenti
    - Almeno uno UDT, l'altro può essere anche built-in
- Pattern obbligatori per << e >>
  - operator <<: un argomento ostream&, ritorna ostream&
  - operator>>: un argomento istream&, ritorna istream&



## Overloading: membro

- Operatori implementati con funzione membro
  - Un argomento in meno
- Il primo argomento è sostituito dall'oggetto su cui è invocata la funzione
  - Lo UDT deve essere il primo argomento
    - int + T non è accettato – lo è solo T + int
    - Meno flessibile!



# Overloading operatore ++

- operator++ ha due implementazioni
  - preincremento
  - postincremento

```
T& operator++(T& t);           // preincremento
T operator++(T& t, int);       // postincremento: argomento
                                // int dummy
```

- Postincremento ritorna una copia del valore precedente (non è lvalue)
  - Può ritornare T o T&
  - Più comunemente ritorna T



# Overloading operatore ++

```
Month operator++(Month& m)
{
    m = (m == Month::dec) ? Month::jan : Month(int(m) + 1);

    return m;
}
```

- operator++ può ritornare Month o Month&
  - Il comportamento è diverso!



# Operatore ternario

```
Month operator++(Month& m)
{
    m = (m == Month::dec) ? Month::jan : Month(int(m) + 1);

    return m;
}
```

- Notare l'operatore "?:" – **operatore ternario**
  - m diventa jan se m == Month::dec, altrimenti Month(int(m) + 1)

Gli operandi sono:

- m = Month::dec
- Month::jan
- Month(int(m) +1)

"m=" non è incluso nel primo operatore.

Il primo operatore viene valutato, il secondo è quello che viene restituito se è true, il terzo se è false.



# Operatore ternario

- ?:
- <condizione> ? <se vero> : <se falso>
- Esempio:

```
int a = 0, b = 4;  
  
int max = a > b ? a : b;
```

- Utile per scritture più compatte
  - Quando le espressioni non sono troppo lunghe



# Overloading operatore <<

```
vector<string> month_tbl;  
// month_tbl inizializzato con le stringhe dei nomi dei mesi  
  
ostream& operator<<(ostream& os, Month& m)  
{  
    return os << month_tbl[int(m)];  
}
```

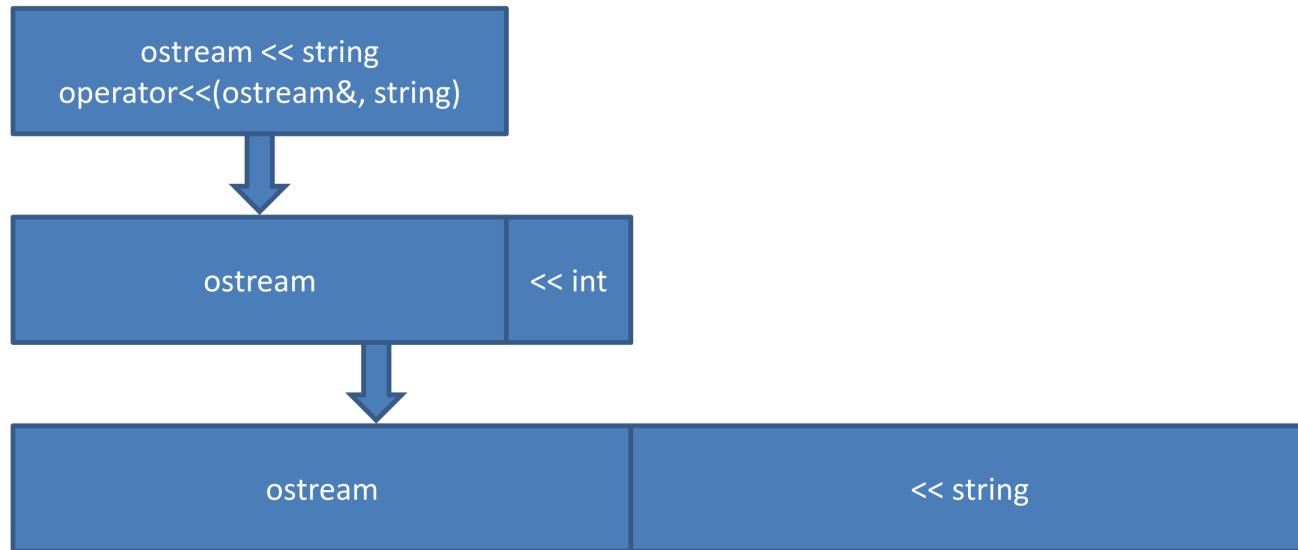
- **operator<< deve ritornare lo stream passato in input**

Il primo argomento **\*deve\*** essere reference a `ostream` (una classe che rappresenta tutti gli oggetti su cui è possibile scrivere).



# Overloading operatore <<

```
int i = 4;  
  
cout << "Il valore " << i << " è stato scritto in i\n";
```



Ci sono tre operatori.

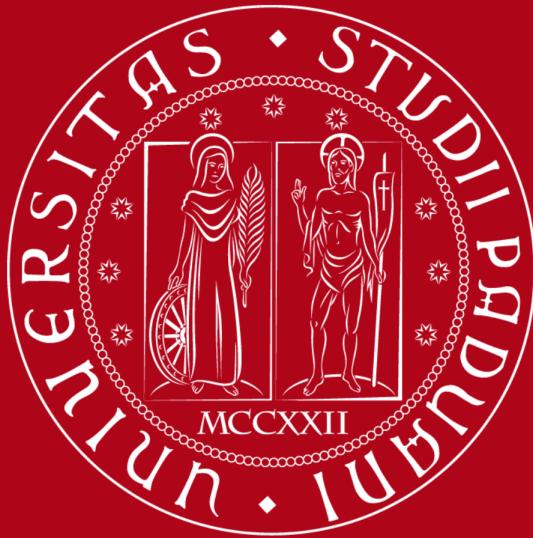
Viene letto da sinistra. Il primo ha come operatore a sinistra ostream, come secondo una stringa. Stampa, poi è costretto a ritornare un ostream, quindi lo prende, lo usa per scriverci e lo restituisce.

Visto che ha restituito uno stream, il secondo operatore lo prende e lo concatena con l'int, che a sua volta lo usa e lo restituisce.

Alla fine, anche il terzo operatore prende ostream e lo restituisce.

In questo modo si possono concatenare più operazioni assieme.

Implementando l'operatore << con una funzione membro, il primo operatore è un oggetto, per questo non è da fare.



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Overloading degli operatori

Stefano Ghidoni



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE