# operatori

- un operatore è analogo a una function
- operatore binario

```
var1 # var2
function # (var1, var2)
```

- operatore unario

```
# var3
function # (var3)
```



# operator overload

- in C++ è possibile definire operatori associati a classi
- definizione come function con:
  - nome - **operator#**
  - numero di argomenti
    - unario = un argomento
    - binario = due argomenti

OperatorOverloadingSyntax.cpp

Fint.cpp, FalseAdd.cpp

# argomenti

- se definiti internamente alla class è implicito il riferimento all'oggetto this come valore sinistro dell'espressione
  - unario = zero argomenti
  - binario = un argomento (valore destro)
- il tipo del secondo argomento consente l'overloading di operatori binari nella stessa classe



# valore ritornato

- dipende dalla semantica dell'operatore
- tipicamente oggetto (o reference a oggetto) della classe
  - consente la costruzione di espressioni
- eccezione tipica : operatori condizionali

# Pre/Post-fix increment

```
// Prefix ++i
const Integer& operator++(Integer& a) {
  a.i++;
  return a;
}
// Postfix i++
const Integer operator++(Integer& a, int) {
  Integer before(a.i);
  a.i++;
  return before;
}
```

`const` per eguagliare i casi di `(++a).F()` *(a)* e `(a++).F()` *(temp)*

OverloadingUnaryOperators.cpp



# esempio overloaded +

Evita uso come lvalue
(a+b) = (c+d)

Previene modifiche

```
const Integer
  operator+(const Integer& left,
            const Integer& right) {
  return Integer(left.i + right.i);
}
```

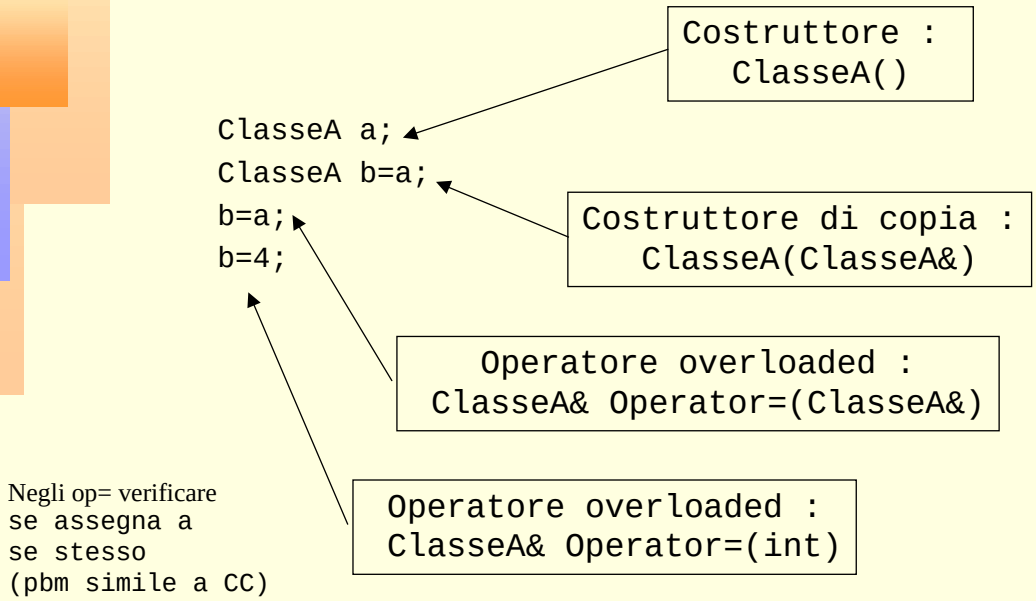
Evita creazione di temporanei:
Integer tmp(left.i+right.i);
return tmp
(costruttore, copy const, distruttore)

# esempio overloaded <<

```
ostream&
operator<<(ostream& os, const IntArray& ia) {
  os << " (" ;
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << ") " ;
  return os;
}
```



# Assegnazione e Costruttori




Negli op= verificare se assegna a se stesso (pbm simile a CC)

# Es: numeri complessi

- il C++ non ha un tipo di dato “numero complesso e neppure le sue operazioni specifiche

**N = ParteReale + *i* ParteImmaginaria**

**A = a1 + *i* a2**
**B = b1 + *i* b2**

**A + B = (a1+b1) + *i* (a2+b2)**



# Classe Complesso

```
class complesso {
   double parte_reale;
   double parte_imm;
public:
   complesso() {
        parte_reale = 0;
        parte_imm = 0;
   }
   complesso(double init_reale, double init_imm) {
        parte_reale = init_reale;
        parte_imm = init_imm;
   }
   double set_reale();
   double set_imm();
   double get_reale();
   double get_imm();
   …
}
```

```
void main ()
…
complesso origine;
complesso punto(3.0,2.0);
…
```

# function operator+

- *funzione friend (NB, può accedere ai membri privati)*

```
friend
complesso operator+ (complesso &oper1,complesso &oper2)
{
  complesso somma(oper1.reale()+oper2.reale(),
                  oper1.imm() + oper2.imm()    );
  return (somma);
}
```



# function operator+=

```
inline complesso& operator += (complesso &oper1,
                               complesso &oper2)
{
  oper1.set_reale( oper1.reale()+ oper2.reale() );
  oper1.set_imm(oper1.imm() + oper2.imm() );
  return (oper1);
}
```

**EQUIVALENTE A (*SOLO SE MEMBRO DELLA CLASSE)***

```
inline complesso& operator += (complesso &oper2)
{
  this->parte_reale += oper2.reale() ;
  this->parte_imm   += oper2.imm() ;
  return (*this);
}
```

this = puntatore che si riferisce all'oggetto corrente
*this = indirizzo dell'oggetto

# meccanica della creazione

- storage
  - statico (compile time)
  - dinamico su stack (block scope)
  - dinamico su heap (new/delete)
- inizializzazione
  - costruttore
  - costruttore di copia



# new /delete

- l'operatore new riserva lo spazio necessario ad un oggetto e ne richiama il costruttore (passa indirizzo)
- l'operatore delete richiama il distruttore e poi libera lo spazio in memoria

```
ClasseA  *aa  = new ClasseA ;
delete *aa ;
```

- overloaded new/delete
  - cambiano solo storage, non costr /distruttori