



Programmazione a oggetti

Costruttori di copia, overloading degli operatori

**A.A. 2020/2021
Francesco Fontanella**

Passaggio di Oggetti a Funzioni

- Gli oggetti possono essere passati alle funzioni come qualsiasi altro tipo di variabile anche per valore:
 - alla funzione viene passata una copia dell'oggetto.
- È quindi necessario creare un nuovo oggetto.
- **Domande**
 1. Quando viene creata la copia viene eseguita la funzione costruttore?
 2. E quando la copia viene distrutta viene eseguita la funzione distruttore?

Risposte

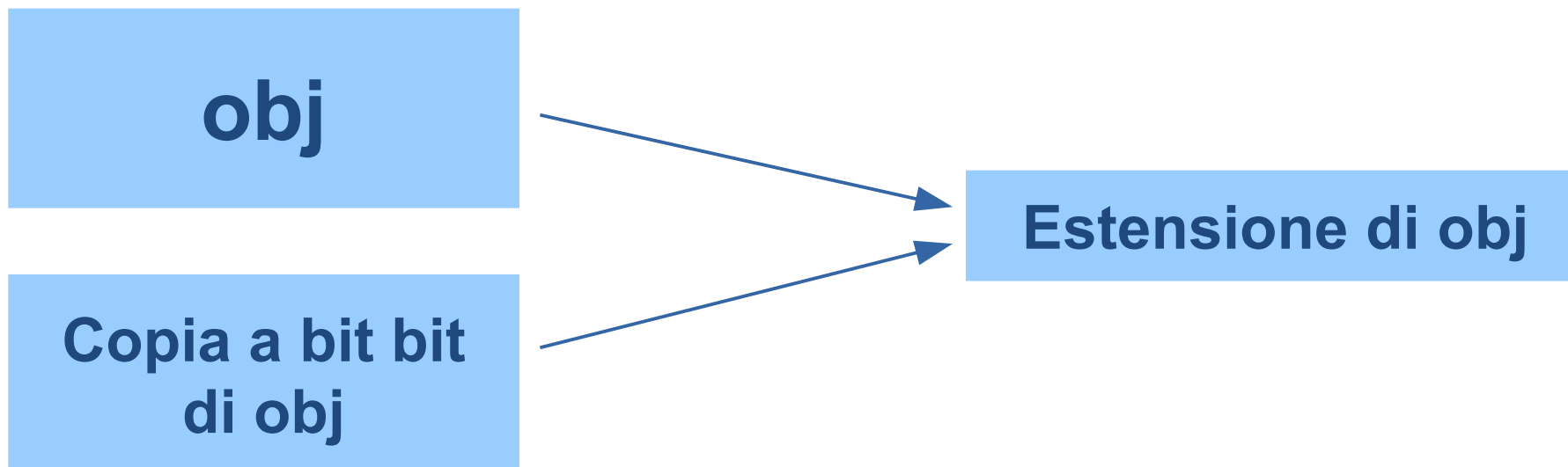


- 1.NO:** quando si passa un oggetto ad una funzione si intende lo stato attuale dell'oggetto. Se venisse richiamato il costruttore sulla copia, lo riporterebbe allo stato iniziale.
- 2.SÌ:** È necessario per distruggere la copia passata alla funzione chiamante
 - 1.Se la copia è costruita bit a bit e questo può creare problemi quando l'oggetto copiato possiede un'estensione

Oggetti con estensione dinamica



- L'operazione di copia di default eseguita dal compilatore è la copia bit a bit
- Questo tipo di copia può creare problemi quando l'oggetto copiato possiede un'estensione dinamica!



Esempio

stack.h

```
Class Stack {  
    public:  
        Stack();  
        ~Stack();  
        .  
        .  
  
    private:  
        int *st_ptr;  
        int num;  
        .  
        .  
};
```

stack.cpp

```
// Costruttore  
Stack::Stack()  
{  
    st_ptr = new int[SIZE];  
    num = 0;  
}  
  
// Distruttore  
Stack::~~Stack()  
{  
    delete [] st_ptr;  
}  
  
    .  
    .
```

Allocazione dinamica
del vettore

Dealloca il vettore

```
#include Stack.h
```

```
void funz(Stack s)
{
    .
    .
}
```

```
main()
{
    Stack s1;

    .
    .
    funz(s1);
```

```
s1.pop();

}
```

**Viene chiamato il costruttore di Stack
che effettua un'allocazione dinamica**

Accadono i seguenti eventi:

1. si costruisce, sullo stack, una copia di s1 per passarla a funz, senza chiamare il costruttore. La copia punterà alla stessa area di memoria heap puntata da s1.
2. al termine della funzione viene chiamato il distruttore sulla copia, ma poiché la copia punta alla stessa area di s1, viene deallocata la memoria puntata da s1

**ERRORE!: il vettore puntato da s1
è stato deallocato dal distruttore della sua copia
passato a funz!**

Restituzione di oggetti

```
#include Stack.h
```

```
Stack funz()  
{  
    Stack s;  
    .  
    .  
    return s;  
}
```

Una funzione può restituire al chiamante un oggetto

```
main()  
{  
    Stack s1;  
  
    s1 = funz();  
  
    return;  
}
```

Questa assegnazione crea una copia bit a bit dell'oggetto locale di funz e la copia in s1.

Dopodichè l'oggetto interno a funz viene distrutto si hanno gli stessi problemi del caso precedente

Il Costruttore di Copia

- Crea un oggetto a partire da un altro oggetto della classe
- È necessario definirlo se e solo se la classe ha un'estensione dinamica
- **Sintassi**

```
class MyClass{
```

```
    .
```

```
    .
```

```
MyClass(const MyClass& s);
```

```
    .
```

```
    .
```

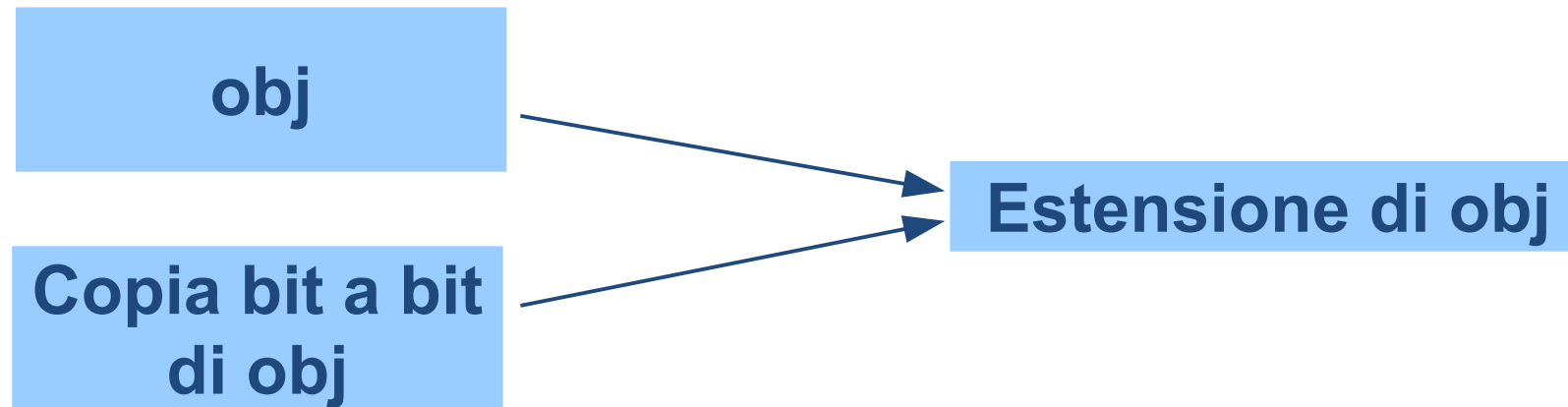
```
};
```

**Impedisce la modifica dell'oggetto
passato per riferimento**

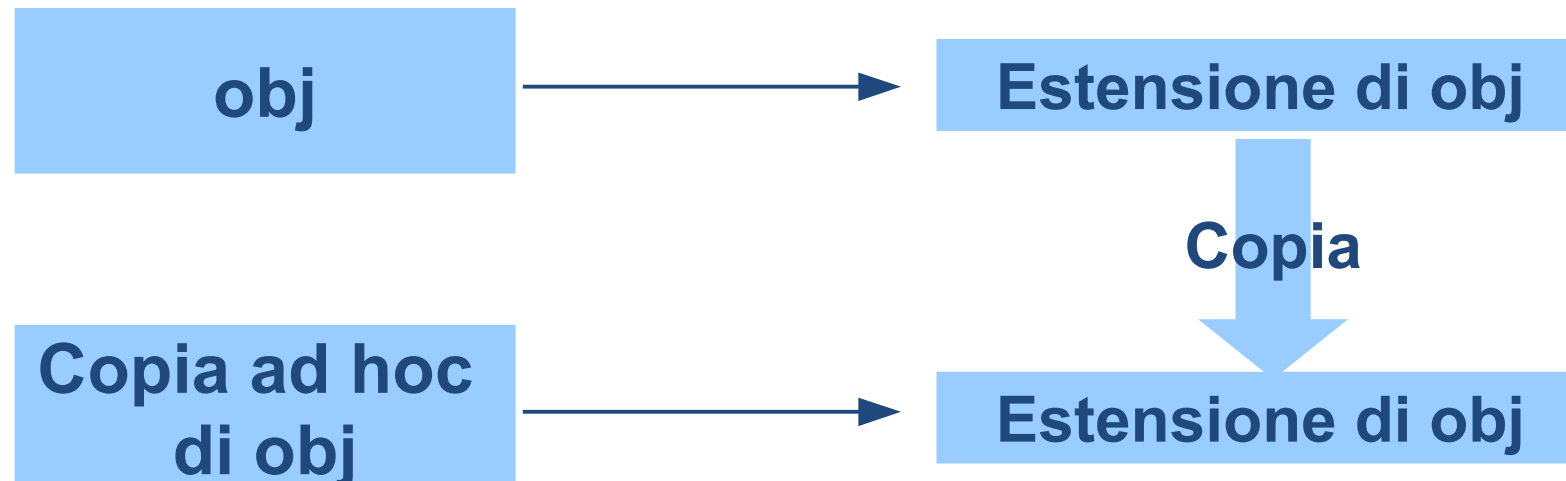
NOTA

L'oggetto da copiare deve essere necessariamente passato per riferimento!!

Costruttore di copia di default (bit a bit)



Costruttore di copia ad hoc



Chiamata dei costruttori di copia



- I costruttori di copia sono chiamati in maniera implicita alla:
 - definizione di un oggetto per inizializzarlo con il valore di un altro oggetto:
`Myclass m2(m1);`
 - chiamata di una funzione per inizializzare un argomento (oggetto) passato per valore.
 - ritorno da una funzione, per restituire un oggetto per valore

Costruttore di Copia: esempio

```
Class Stack {  
    public:  
        Stack(int dim = STACK_SIZE);  
        ~Stack();  
        Stack(const Stack& s);  
        .  
        .  
        .  
  
    private:  
        TipoValue *v; // array DINAMICO per la memorizzazione  
        int last;  
        int len;  
};
```

parametro di default
per il costruttore



```
// Costruttore:  
// alloca un array con una dimensione di default  
Stack::Stack(int dim)  
{  
    v = new TipoValue[dim];  
    last = -1;  
    len = dim;  
}  
  
// Distruttore  
Stack::~~Stack()  
{  
    delete [] v;  
}
```

```
// Costruttore di copia
Stack::Stack(const Stack &s)
{
    int i;

    last = s.last;
    len = s.len;

    // Si alloca spazio per il vettore e se ne fa la copia
    v = new TipoValue[len];

    for (i=0; i < last; ++i)
        v[i] = s.v[i];
}
```

Riassumendo...

- Se la nostra classe contiene puntatori che fanno riferimento ad estensioni dinamiche è **NECESSARIO** definire, oltre al costruttore/i:
 - Costruttore di copia
 - Operatore di assegnazione (lo vedremo poi)
 - Distruttore
- In caso contrario, le operazioni di cui sopra **NON** sono necessarie.

Esempio

```
Class Stack {  
    public:  
        Stack();  
        .  
        .  
  
    private:  
        TipoValue v[VEC_SIZE]; // array STATICO per la memorizzazione  
        int last;  
        int len;  
};
```

```
// Costruttore  
Stack::Stack(int dim)  
{  
  
    last = -1;  
    len = dim;  
}
```

Accesso alle variabili membro: lettura



- Per accedere a variabili **private** è necessario definire delle funzioni;
- Una tipica definizione di funzione di accesso è del tipo:
TipoValue getValue(){**return** value;}
- Tipicamente queste funzioni sono definite nella dichiarazione della classe:

```
Class Stack {  
    public:  
        .  
        .  
        int getNum(){return num;}  
    private:  
        int num;  
        .  
        .  
};
```

Variabili di tipo stringa (alla C)

■ Soluzione 1

si passa un parametro in cui copiare il valore:

```
void getString(char *str){ strcpy(str, string);}
```

■ Soluzione 2

si restituisce un puntatore ad una stringa allocata dinamicamente:

```
char *C::getString()  
{  
    char *str;  
  
    str = new char[MAX_STRING];  
    strcpy(str, string);  
  
    return str;  
}
```


Soluzione 1: Esempio



```
Class Persona{
    // Funzioni di accesso
    void getNome(char *n){strcpy(n, nome);}
    void getCognome(char *c){strcpy(c, cognome);}
    .
    .

private:
    char nome[MAX_STRING];
    char cognome[MAX_STRING];
    .
    .
};
```

```
main()
{
    char s1[MAX_STRING], s2[MAX_STRING];
    Persona p;
    .
    .

    p.getNome(s1); // accesso al nome dell'oggetto p
    p.getCognome(s2); // accesso al cognome dell'oggetto p
}
```

Soluzione 2: Esempio

```
Class Persona{
    // Funzioni di accesso
    char* getNome();
    char* getCognome();
    .
    .

private:
    char nome[MAX_STRING];
    char cognome[MAX_STRING];
    .
    .
};
```

```
main()
{
    char *s1, *s2;
    Persona p;
    .
    .

    s1 = p.getNome(); // accesso al nome di p
    s2 = p.getCognome(); // accesso al cognome di p
}
```

```
char* Persona::getNome()
{
    char *str;

    str = new char[MAX_STRING];
    strcpy(str, nome);

    return str;
}

char* Persona::getCognome()
{
    char *str;

    str = new char[MAX_STRING];
    strcpy(str, cognome);

    return str;
}
```

Modifica delle variabili membro

- In alcuni casi è necessario definire anche delle funzioni che consentano la modifica (dall'esterno) delle variabili di un oggetto.
- Una tipica definizione di funzione di modifica, effettua dei controlli per verificare la correttezza del valore da assegnare
- **Esempio**

```
void MyClass::setVar(TipoValue val)
{
    if (val >= MINVAR && val <= MAXVAR)
        var = val;
    else cout<<"ERRORE: valore val errato!";
}
```

Osservazioni

- La scelta di nomi come `getNomevar` e `setNomevar` è una buona norma di programmazione (best practice), ma non è assolutamente prescritta dal linguaggio C++
- Le funzioni (sia `get` che `set`) vanno definite SOLO per le variabili che devono essere accedute dall'esterno

Esempio



```
Class MyClass {  
    public:  
        // Funzioni Costruttore  
        MyClass();  
        .  
        .  
  
        // Funzioni di accesso  
  
        int getN(){return N;} //restituisce il valore di N  
        char getCh(){return ch;} //restituisce il valore di ch  
        // Funzioni di modifica  
        void setCh(char c){char = c;} // assegna valore a ch  
  
    private:  
        int N;  
        char ch;  
        float x;  
};
```

In questo esempio, solo la variabile **ch** è modificabile dall'esterno

Le funzioni inline

- il compilatore copia il codice della funzione in ogni punto in cui essa viene invocata (come se fosse una macro)
- il programma verrà così eseguito più velocemente perché non si dovrà eseguire il codice associato alla chiamata alla funzione
- per creare una funzione inline si deve inserire la parola riservata **inline** all'inizio dell'intestazione
- **Esempio**

```
inline int MyClass::funz()  
{  
    // Implementazione della funzione  
    .  
    .  
}
```

Osservazioni

- Le funzioni inline:
 - sono convenienti quando la funzione è chiamata spesso ed il suo codice è breve
 - aumentano però le dimensioni dell'eseguibile
- Le funzioni membro definite nella dichiarazione vengono automaticamente trasformate dal compilatore in funzioni **inline** all'inizio dell'intestazione

- **Esempio**

```
Class MyClass {  
    public:  
        .  
        .  
  
    int getN(){return N;}  
    char getCh(){return ch;}  
  
    void setCh(char c){char = c;}  
  
        .  
        .  
};
```

inline

Accesso ai membri di una classe

Accesso ai membri di una classe

- In ogni funzione membro è possibile fare riferimento alle variabili della classe senza nessuna ambiguità.

```
Class MyClass {  
    public:  
        MyClass();  
        .  
        .  
        funz(Tipo val);  
        .  
        .  
    private:  
        Tipo1 var1;  
        Tipo1 var2;  
};
```

```
Myclass::funz(Tipo val)  
{  
    var2 = pow(val, 2);  
}
```

```
main()  
{  
    MyClass m1, m2;  
  
    m1.funz(2); // chiamata sull'oggetto m1  
    m2.funz(5); // chiamata sull'oggetto m2  
}
```

Domanda

qual è il meccanismo che consente alla funzione membro di individuare le variabili specifiche dell'oggetto sul quale la funzione è stata chiamata?

Il puntatore **this**

- Nella dichiarazione della classe, per ogni funzione membro il preprocessore introduce, in maniera automatica, un parametro nascosto: l'indirizzo dell'oggetto a cui applicare la funzione
- Questo parametro è il puntatore di tipo costante **this**.
- In pratica, il puntatore **this** consente di identificare l'oggetto al quale applicare una certa funzione della classe

myclass.h

```
Class MyClass {  
    .  
    .  
    Tipo funz(Tipo val);  
    .  
    .  
};
```

PREPROCESSORE

myclass.h modificato

```
Class MyClass {  
    .  
    .  
    Tipo funz(Myclass* const this, Tipo val);  
    .  
    .  
};
```

- La trasformazione interessa anche i file .cpp:

myclass.cpp

```
Tipo MyClass::funz(Tipo1 val)
{
    var1 = pow(2, val);
};
```

PREPROCESSORE

myclass.cpp modificato

```
Tipo funz(Myclass* const this, Tipo1 val)
{
    this->var1 = pow(2, val);
};
```

- La trasformazione interessa anche tutte le chiamate delle funzioni della classe:

```
main ()
{
    Myclass m, *mp;
    Tipo1 x;
    .
    .
    m.funz(x);
    .
    .
    mp->funz(x);
}
```

PREPROCESSORE

```
main ()
{
    Myclass m;
    Tipo1 x;
    .
    .
    funz(&m, x);
    .
    .
    funz(mp, x);
}
```



Variabili e funzioni static

Variabili static

- Se una variabile membro è dichiarata **static**, il compilatore ne crea una sola copia, condivisa da tutte le istanze di quella classe
- consentono la condivisione di informazione tra istanze della stessa classe
- le variabili **static**:
 - Sono inizializzate a zero
 - Non occupano spazio di memoria all'interno delle istanze
 - devono essere definite come variabili globali.

Esempi

```
class ShareVar {  
    static int num;  
public:  
    void setNum(int i) { num = i; };  
    void showNum() { cout << num << " " << endl; }  
};
```

```
int ShareVar::num; // definisce num come variabile globale
```

```
int main()  
{  
    ShareVar a, b;  
  
    a.showNum(); // visualizza 0  
    b.showNum(); // visualizza 0  
  
    a.setNum(10); // imposta static num a 10  
  
    a.showNum(); // visualizza 10  
    b.showNum(); // anche questa istruzione visualizza 10  
  
    return 0;  
}
```

OUTPUT

```
0  
0  
10  
10
```


■ Esempio d'uso

- contare il numero di oggetti istanziati di una certa classe:

main.cpp

```
int MyClass::count;
int main()
{
    MyClass m, MyClass_array[50], *mp;

    cout<<endl<<"oggetti esistenti: "<<m.getCount();

    // Alloco memoria per altre 50 istanze
    mp = new MyClass[50];
    cout<<endl<<"oggetti esistenti: "<<m.getCount();

    // dealloco...
    delete [] mp;
    cout<<endl<<"oggetti esistenti: "<<m.getCount();

    return 0;
}
```

counter.h

```
class MyClass {
    static int count;
public:
    MyClass(){++count;} // costruttore
    ~MyClass(){--count;} // distruttore
    int getCount(){return count;}
};
```

OUTPUT

```
oggetti esistenti: 51
oggetti esistenti: 101
oggetti esistenti: 51
```

Funzioni *static*

- Possono accedere solo ai membri static della classe
- Possono essere chiamate anche se non esistono oggetti della classe
- Di solito usate per accedere (e inizializzare) le variabili static della classe
- **Esempio**

```
int main()
{
    .
    .

    cout<<endl<<"oggetti esistenti: "<<Myclass::get_count();

    .
    .

    return 0;
}
```

```
class MyClass {
    static int count;
public:
    MyClass(){++count;}
    ~Myclass(){--count;}
    static int get_count(){return count;}
```

funzioni (e classi) friend

- Funzioni esterne alla classe che possono accedere ai suoi membri privati (incapsulamento più flessibile)
- Utili quando due o più classi contengono membri correlati con altre parti del programma.
- Anche tra classi:
 - tutte le funzioni della classe friend avranno accesso ai membri privati della classe.
- La “friendness” non è automaticamente reciproca:
 - A friend di B NON implica B friend di A

Esempio

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

```
main()  
{  
    MyClass m;  
  
    m.set_ab(2, 4);  
  
    cout<<sum(m);  
}
```

```
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;  
}
```

sum **non è membro di** MyClass,
ma **PUÒ** accedere ai suoi membri
privati perché dichiarata friend

Overloading degli operatori

Overloading degli operatori

- Consente di adattare gli operatori per svolgere operazioni specifiche di una classe
- È possibile per la maggior parte degli operatori:
 - $+$, $-$, $*$, $/$, ecc.
- Realizzato per mezzo delle funzioni operator, che possono essere:
 - membro della classe;
 - esterne dichiarate friend per la classe.

Funzioni operator membro

■ forma generale:

```
Tipo nome-classe::operator#(Tipo1 arg1, Tipo2  
    arg2, ...)  
{  
    // istruzioni  
    .  
    .  
    .  
}
```

NOTA

il simbolo # rappresenta il generico operatore da sovraccaricare

Esempio

```
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);  
};
```

Costruttore con parametri di default


```
Complex 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;  
}
```

**DEVE restituire un oggetto
della classe Complex**

```
main()
{
    Complex c1, c2(1,1), c3(4,5);

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

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

`c1 = c2 + c3;`

preprocessore

output

```
re: 0 im: 0
re: 1 im: 1
re: 4 im: 5
re: 5 im: 6
```

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

`c1 = operator+(&c2, c3);`

NOTA

Per gli operatori binari è sempre l'oggetto di sinistra a generare la chiamata a `operator+`

Regole

- È possibile modificare il significato di un operatore esistente, non è possibile creare nuovi operatori
- Non è opportuno ridefinire la semantica di un operatore applicato a tipi predefiniti.
- Non è possibile cambiare precedenza, associatività e “arity” (numero di operandi)
- Non è possibile usare argomenti di default
- in analogia con gli operatori standard, è opportuno definire sempre degli operatori che non modificano gli operandi

Operatore sottrazione

```
Complex Complex::operator-(Complex op2)
{
    Complex tmp;

    tmp.re = re - op2.re;
    tmp.im = im - op2.im;

    return tmp;
}
```

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

Operatori incremento

prefisso

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

postfisso

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

Così possono essere usati in espressioni
del tipo; `c1=++c2` oppure `c1=c2++`

NOTA

Per distinguere la definizione dell'operatore postfisso da quella dell'operatore prefisso è necessario usare un parametro fittizio di tipo int

```
main()
{
    Complex c1(1,2), c2(3,5), c3;

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

    ++c1; // operatore prefisso
    c3 = c2++; // operatore postfisso
    c1.show();

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

    c1 = c2 - c3;

    c1.show();
    (c1+c2).show();
}
```

output

```
re: 1 im: 2
re: 3 im: 5
re: 2 im: 3
re: 3 im: 4
re: 3 im: 4
re: -1 im: -2
re: 2 im 2
```

Domanda: su quale oggetto viene chiamata la funzione show()?

Operatore di assegnazione

- Default: copia bit a bit:
- Overloading necessario per le classi che hanno un'estensione dinamica.
- Ha la forma:

C& operator=(const C &ob)

- Deve consentire assegnazioni multiple

a=b=c . . .

Esempio



```
class MyClass{
    int n;        // cardinalità dell'array
    int *v;       // array allocato dinamicamente di
                  // cardinalità n
    .
    .
    public:
        MyClass();
        .
        .
        MyClass& operator=(const MyClass &other);
};
```


restituzione per riferimento



Passaggio per riferimento
più efficiente)



```
Myclass& Myclass::operator=(const Myclass &other)
{
    int i;
    if (this != &other) { // Assegnazione a se stesso?
        delete [] v; // si dealloca il vecchio array

        n = other.n; // si aggiorna la cardinalità
        v = new int [n]; // si alloca il nuovo array

        // si copia il vettore
        for (i=0; i < n; ++i)
            v[i] = other.v[i];
    }

    return *this;
}
```

rende possibile assegnazioni multiple
del tipo: `c1 = c2 = c3;`

```
class MyClass{
    int i;
public:
    MyClass(){cout<<endl<<"COSTRUTTORE: "<<this;};
    MyClass(const MyClass& o){cout<<endl<<"COPIA: "<<this;}
    MyClass operator=(const MyClass &m){
        cout<<endl<<"ASSEGNAZIONE: "<<this;
        return *this;
    }
};
```

```
int main() {
    MyClass m1, m2, m3;

    m1 = m2 = m3;

    return 0;
}
```

OUTPUT

```
COSTRUTTORE:0x7ffff81beac0 m1
COSTRUTTORE:0x7ffff81bead0 m2
COSTRUTTORE:0x7ffff81beae0 m3
ASSEGNAZIONE:0x7ffff81bead0 m2
COPIA:0x7ffff81beaf0 ?
ASSEGNAZIONE:0x7ffff81beac0 m1
COPIA:0x7ffff81beab0 ?
```

m1.operator=(m2.operator=(m3));

```
class MyClass{  
    . // come prima...  
    .  
    MyClass& operator=(const MyClass &m){  
        cout<<endl<<"ASSEGNAZIONE: "<<this;  
        return *this;  
    }  
};
```

**restituzione
per riferimento
(return by reference)**

```
int main() {  
    MyClass m1, m2, m3;  
  
    m1 = m2 = m3;  
  
    return 0;  
}
```

OUTPUT

```
COSTRUTTORE:0x7fffc7fb2920 m1  
COSTRUTTORE:0x7fffc7fb2930 m2  
COSTRUTTORE:0x7fffc7fb2940 m3  
ASSEGNAZIONE:0x7fffc7fb2930 m2  
ASSEGNAZIONE:0x7fffc7fb2920 m1
```

m1.operator=(m2.operator=(m3));

Forme abbreviate

- È possibile effettuare anche l'overloading delle forme abbreviate degli operatori, tipo: `+=`, `*=`, `-=` ecc.

- **Esempio**

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

    return *this;
}
```



```
main
{
    Complex c1, c2;

    .
    .
    .
    c1 += c2;

}
```

Overloading con funzioni friend

- L'overloading può essere eseguito anche per mezzo di funzioni **friend** non membro della classe in esame.

Esempio

```
class Complex {  
    :  
    :  
    :  
    friend Complex operator+(Complex op1, Complex op2);  
    friend Complex operator++(Complex op);  
};
```

NOTA

In questo caso, il #argomenti coincide con #operandi.

```
Complex operator+(Complex op1, Complex op2)  
{  
    Complex tmp;  
  
    tmp.re = op1.re + op2.re;  
    tmp.im = op1.im + op2.im;  
  
    return tmp;  
}
```

```
Complex operator++(Complex &op)  
{  
    ++op.re;  
    ++op.im;  
  
    return *this;  
}
```

- Di solito le funzioni friend sono da preferire, ci sono però situazioni in cui le funzioni friend sono preferibili...

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

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

    return tmp;
}
```

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

```
int main()
{
    Complex c1, c2;

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

Soluzione



- L'operatore visto in precedenza può essere reso più flessibile utilizzando due funzioni esterne, dichiarate **friend** per la classe:

```
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;
}
```

```
int main()
{
    Complex c1, c2;

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


Overloading operatori di I/O

- In C++ l'overloading delle operazioni di I/O può essere fatto SOLO per mezzo di funzione esterne dichiarate **friend**
- Questo perchè nelle operazioni di I/O lo stream appare sempre a sinistra
- **Esempio**

```
class Complex {  
    public:  
  
        .  
        .  
    private:  
        float re, im;
```

```
    // Inseritore della classe  
    friend ostream& operator<<(ostream &os, Complex C);  
    // Estrattore della classe  
    friend istream& operator>>(istream &in, Complex &C);  
};
```

gli stream vanno **SEMPRE**
passati per riferimento

inseritore

```
ostream& operator<<(ostream &os, Complex op)
{
    os << op.re;

    if (im > 0)
        os<<" +";
    else if (im < 0)
        os<<" ";
    else return os;
    os<<op.im<<"i";

    return os;
}
```

estrattore

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


    in >> tmp.re;
    in >> tmp.im;
    op = tmp;


    return in;
}
```

Passaggio per
riferimento

```
int main()
{
    Complex c1, c2, c3;

    cout << "\n inserisci il primo operando: ";
    cin >> c1;
    cout << "\n inserisci il secondo operando: ";
    cin >> c2;
    c3 = c1 + c2;
    cout << c3;
    cout << "\n";
}
```

cout << c3;  **equivale a** **operator<<(cout, c3);**

cin >> c1;  **equivale a** **operator>>(cin, c3);**

Esempio

```
class Studente {  
    char nome[MAX_STRING];  
    char cognome[MAX_STRING];  
    int  matr;  
public:  
    void input();    // input da utente  
    void output();   // output su schermo  
  
    // I/O su stream  
    friend ostream& operator<<(ostream &os, Studente s);  
    friend istream& operator>>(istream &in, Studente &s);  
}
```

```
ostream& operator<<(ostream &os, studente &s)
{
    os<<endl;
    os<<nome<<" , ";
    os<<cognome<<" , ";
    os<<matr;

    return os;

}
```



```
istream& operator>>(istream& in, Studente &s)
{
    char str[MAX_LINE];

    if (in.getline(str, MAX_LINE, ','))
        strcpy(nome, str);
    else return in;

    if (in.getline(str, MAX_LINE, ','))
        strcpy(cognome, str);
    else return in;

    if (in.getline(str, MAX_LINE)
        matr = atoi(str);
    else return in;

    return in;
}
```

Leggere e scrivere array su file

```
void read_students(istream &in, Studente s[], int &n)
{
    Studente tmp;
    n = 0;
    while(in>>tmp)
        s[n++] = tmp;
}
```

Domanda

È possibile implementare questa funzione senza usare la variabile tmp?



```
void write_students(istream &out, Studente s[], int n)
{
    int i;

    i = 0;
    while(out<<s[i] && i < n)
        ++i;

    if (i < n)
        out<<endl<<"ERRORE!: impossibile scrivere l'array!";

    return;
}
```


Osservazioni

- Definiti gli operatori di I/O, le funzioni precedenti possono essere generalizzate per qualsiasi classe (Tipo):

```
void read_vec(istream &in, TipoValue v[], int &n)
{
    TipoValue tmp;
    n = 0;
    while(in>>tmp)
        v[n++] = tmp;
}
```

```
void write_vec(istream &out, TipoValue v[], int n)
{
    int i;

    i = 0;
    while(out<<v[i] && i < n)
        ++i;

    if (i < n)
        cout<<endl<<"ERRORE!: impossibile scrivere l'array!";

    return;
}
```