Corso di Programmazione a oggetti

Costruttori di copia, funzioni static e overloading degli operatori

a.a. 2014/2015
Francesco Fontanella

Multipli del byte

Multipli del byte					
Prefissi SI			Prefissi binari		
Nome	Simbolo	Multiplo	Nome	Simbolo	Multiplo
kilobyte	kB	10^3	kibibyte	KiB	2^{10}
megabyte	MB	10^6	mebibyte	MiB	2^{20}
gigabyte	GB	10^9	gibibyte	GiB	2^{30}
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}
zettabyte	ZB	10^{21}	zebibyte	*ZiB*	2^{70}
yottabyte	YB	10^{24}	yobibyte	*YiB*	2^{80}

Per ulteriori informazioni:

http://it.wikipedia.org/wiki/Megabyte

Passaggio di Oggetti a Funzioni

- Gli oggetti possono essere passati alle funzioni come qualsiasi altro tipo di variabile.
- Gli oggetti possono essere passati per valore: alla funzione viene passata una copia dell'oggetto.
- È quindi necessario creare un nuovo oggetto.

Passaggio di Oggetti a Funzioni

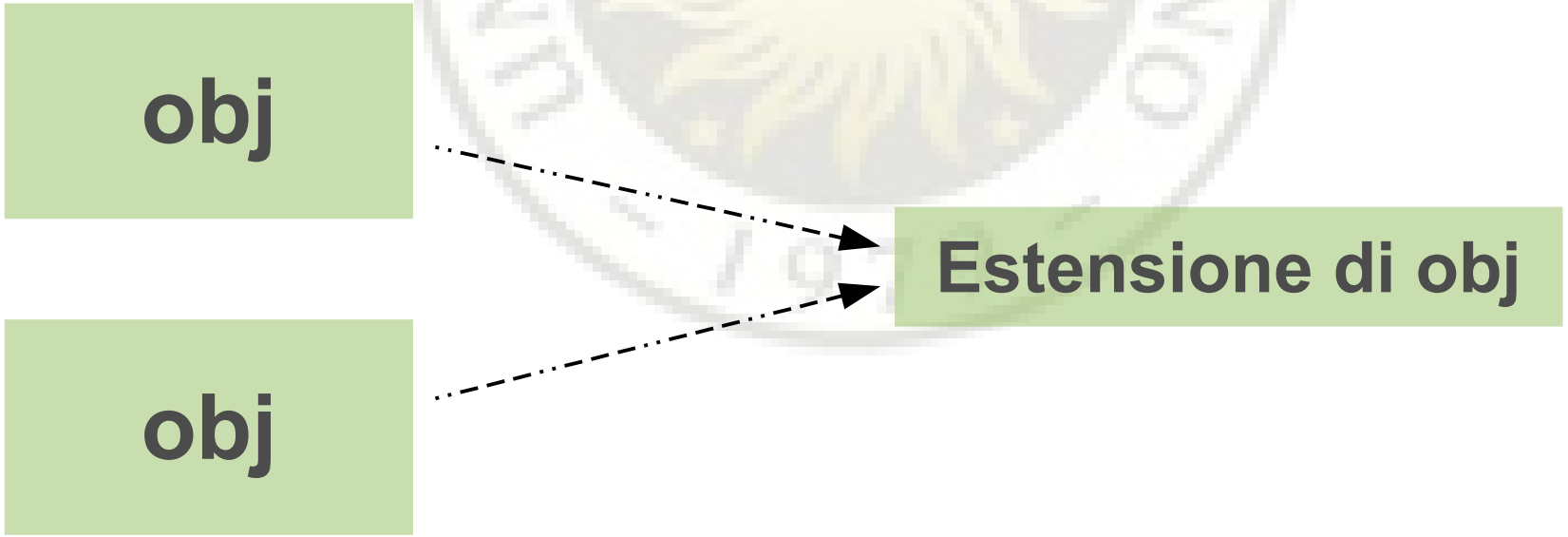
- **Domande**
  1. Quando viene creata la copia viene eseguita la funzione costruttore?
  2. E quando la copia viene distrutta viene eseguita la funzione distruttore?

Risposta 1

- Quando viene costruita la copia per passarla alla funzione NON viene chiamato il costruttore.
- Il motivo è semplice: quando si passa un oggetto ad una funzione si intende lo stato corrente dell'oggetto. Se venisse richiamato il costruttore sulla copia, lo riporterebbe allo stato iniziale.

Risposta 2

- È necessario invece richiamare il distruttore nel momento in cui deve essere distrutta la copia.
- Notiamo che la copia è costruita bit a bit e questo può creare problemi quando l'oggetto copiato possiede un'estensione



```
#include Stack.h

void funz(Stack s)
{
  .
  .
}

main()
{
  Stack s1;

   .
   .
  funz(s1);

s1.pop();

}
```

Viene chiamato il costruttore di Stack che effettua un'allocazione dinamicamente

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;
}

main()
{
  Stack s1;

  s1 = funz();

  return;
}
```

Una funzione può restituire al chiamante un oggetto

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

- La sintassi dei costruttori di copia è la seguente:

Il qualificatore **const** impedisce
la modifica dell'oggetto passato per riferimento

```
class Myclass{
    .
    .
Myclass(const Myclass& s);
    .
    .
};
```

Chiamata dei costruttori di copia

- I costruttori di copia sono chiamati in maniera implicita:
  - All'atto della definizione di un oggetto per inizializzarlo con il valore di un altro oggetto:
    ```
    Myclass m2(m1);
    ```
- All'atto della chiamata di una funzione per inizializzare un argomento (oggetto) passato per valore.

- All’atto del ritorno da una funzione, per restituire un oggetto per valore

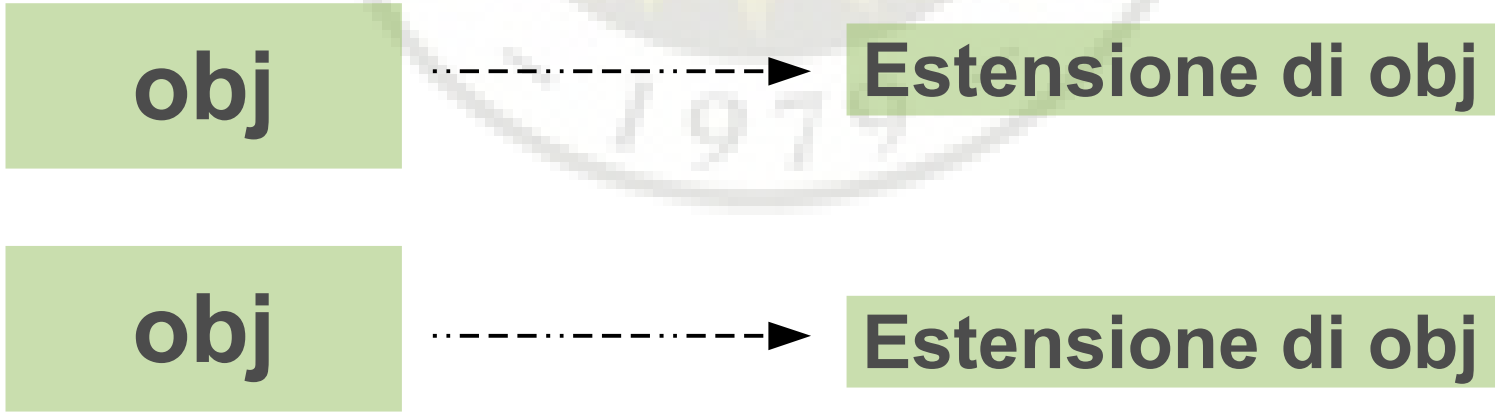
- La definzione di un costruttore di copia consente di risolvere tutti i problemi relativi appunto alla copia di oggetti che hanno un'estensione.

Costruttore di copia di default

- In mancanza della definizione del costruttore di copia, il compilatore richiama un costruttore di copia, detto *di default*
- Il costruttore di copia di default esegue una *copia* bit a bit, limitata alla sola parte base
- Nel caso esista una estensione dinamica il costruttore di copia deve essere esplicitamente definito dal progettista della classe

Costrutture di copia di default (bit a bit)

Costrutture di copia ad hoc



Costruttore di Copia: esempio

```
Class Stack {
  public:
    Stack();
    Stack(int dim);
    ~Stack();
    Stack(const Stack& s);
      .
      .
      .

  private:
    TipoValue *v; // array per la memorizzazione
    int last;     // punta all'ultimo elemento inserito
    int len:      // cardinalità dell'array
};
```

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

// Costruttore 2:
// alloca un array di dimensione dim
Stack::Stack(int dim)
{
 v = new TipoValue[dim];
 last = -1;
 len = dim;
}
```

```
// 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];
}

// Distruttore
Stack::~Stack()
{
  delete [] v;

}
```

Riassumendo...

- Se la nostra classe contiene puntatori che fanno riferimento ad estensioni dinamiche conviene sempre scrivere accuratamente
  - Costruttore
  - Costruttore di copia
  - Operatore di assegnamento (lo vedremo poi)
  - Distruttore

Funzioni di accesso alle variabili membro: lettura

- Lo stato di un oggetto è determinato dal valore assunto dalle sue variabili
- È buona norma della programmazione OO limitare l'accesso a queste variabili definendole **private**
- Nasce quindi l'esigenza di definire delle funzioni che consentano l'accesso a queste variabili

- Una tipica definizione di funzione di accesso è del tipo:

```
Tipo get_value(){return value;}
```

- Tipicamente queste funzioni sono definite all'interno della dichiarazione della classe:

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

Accesso a variabili di tipo stringa

- **Soluzione 1**

  si passa un parametro in cui copiare il valore:

```
void get_value(char *str){ strcpy(str, value);}
```

- **Soluzione 2**

  si restiuisce un puntatore ad una stringa allocata dinamicamente:

```
char *C::get_value()
{
  char *str;

  str = new char[max_string];
  strcpy(str, value);

  return str;
}
```

Soluzione 1: Esempio

```
Class Persona{
 // Funzioni di accesso
 void get_nome(char *n){strcpy(n, nome);}
 void get_cognome(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.get_nome(s1); // accesso al nome dell'oggetto p
 p.get_cognome(s2); // accesso al cognome dell'oggetto p
}
```

Soluzione 2: Esempio

```
Class Persona{
 // Funzioni di accesso
 char* get_nome();
 char* get_cognome();
      .
      .

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

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

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

    return str;
}

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

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

    return str;
}
```

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

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

Modifica delle variabili membro

- Oltre che leggere lo stato di un oggetto è anche necessario modificarlo.
- Pertanto bisogna definire anche delle funzioni che consentano la modifica delle variabili di un oggetto.

- Una tipica definizione di funzione di modifica è del tipo:

```
void set_value(Tipo val){var = val;}
```

- **NOTA**

  La scelta di nomi come get_nomevar e set_nomevar è una buona norma di programmazione, ma non è assolutamente prescritta dal linguaggio C++

Esempio

```
Class Myclass {
  public:
     // Funzioni Costruttore
    Myclass();
         .
         .
    // Funzioni di accesso
   int get_N(){return N;}; //restituisce il valore di N
    char get_ch(){return ch;}; //restituisce il valore di ch
      // Funzioni di modifica
    void set_N(int val){N = val;}; // assegna valore a N
    void set_ch(char c){char = c;}; // assegna valore a ch

  private:
    int N;
    char ch;
    float x;
};
```

Le funzioni membro definite nella dichiarazione vengono automaticamente trasformate dal compilatore in funzioni inline

Le funzioni inline

- il compilatore ricopia realmente il codice della funzione in ogni punto in cui essa viene invocata
- il programma verrà così eseguito più velocemente perché non si dovrà eseguire il codice associato alla chiamata alla funzione
- per creare una funzione ilinea si deve inserire la parola riservata inline all'inizio dell'intestazione:

```
inline int somma10(int n) {return (n+10);}
```

- Le funzioni inline aumentano la velocità del programma e sono convenienti quando la funzione si richiama parecchie volte nel programma e il suo codice è breve

- Le funzioni inline aumentano però le dimensioni dell'eseguibile, poichè il codice della funzione è ripetuto ogni volta che la funzione è invocata\

Accesso ai membri di una classe

Accesso ai membri di una classe

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

- **Domanda**

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

```
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); // modifica le variabili di m1
 m2.funz(5); // modifica le variabili di m2
}
```

Meccanismo di accesso ai membri di una classe

- Nella chiamata di una funzione membro funz il compilatore introduce un primo parametro nascosto che è l'indirizzo dell'oggetto proprio a cui funz viene applicata.

- Tale parametro e' un puntatore costante il cui nome è **this.**

Myclass.h

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

Myclass.h modificato

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

};
```

Il puntatore This

- Consente di identificare l'oggetto istanziato al quale applicare una certa funzione della classe.
- Quando si definisce una funzione membro non si specifica in alcun modo l'oggetto sul quale la funzione verrà chiamata.
- Questa operazione viene fatta in maniera automatica dal preprocessore. Il quale, modifica i riferimenti alle variabili di classe aggiungendo il puntatore this.

- La traformazione interessa anche il file .cpp:

Myclass.cpp

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

};
```

Myclass.cpp modificato

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


};
```

Chiamata delle funzioni membro

- L'istruzione di chiamata di una funzione membro su una specifica istanza di una classe viene anch'essa trasformata dal preprocessore.

- In pratica l'indirizzo dell'oggetto (tramite l'operatore &) viene ricopiato in this e attraverso tale puntatore la funzione opera sull'oggetto proprio:

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

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

}
```


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

}
```

Variabili e funzioni static

Variabili static

- I membri di una classe possono essere dichiarati ***static***.

- Quando la dichiarazione di una variabile membro è preceduta dalla parola **static**, si chiede al compilatore di creare una sola copia di quella variabile per tutti gli oggetti della classe.

- Questo significa che tutte le istanze di quella classe utilizeranno la stessa variabile.

- Tutte le variabili static vengono inizializzate a zero nel momento in cui viene creato il primo oggetto.
- Per le variabili static non si alloca spazio di memoria all'interno delle istanze della classe
- Le variabili static di una classe devono essere definite come variabili globali.
- Le variabili static consentono la condivisione di informazione tra istanze della stessa classe

```
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
```

- Le variabili di tipo static possono essere usate per contare il numero di oggetti istanziati di un certa classe:

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

};
```

SEGUE ...

```
int main()
{
  Counter c, count_array[50], *cp;

  cout<<endl<<"oggetti esistenti: "<<c.get_count();

  // Alloco memoria per altre  50 istanze
  cp = new Counter[50];
  cout<<endl<<"oggetti esistenti: "<<c.get_count();

  // dealloco...
  delete [] cp;
  cout<<endl<<"oggetti esistenti: "<<c.get_count();


  return 0;

}
```

OUTPUT

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

- Le variabili static possono essere usate per condividere risorse comuni a tutti gli oggetti di una classe:

```
class Myclass {
static int resource;
public:
  Myclass();
    .
    .
  bool get_resource();
  void free_resource(){ resource = 0; };
};

bool Myclass::get_resource()
{
 if (resource == 0) {
   resource = 1;
  return true;
 } else return false;
}
```

```
main()
{
  Myclass m1, m2;

        .
        .
        .

  if (m1.get_resource())
    cout<<" la risorsa è di m1";

    if (!m2.get_resource())
    cout<<" m2 non può utilizzare la risorsa";

}
```

Funzioni *static*

- Anche le funzioni membro di una classe possono essere dichiarate *static*
- Vi sono però delle restrizioni:
  - possono accedere solo ai membri static della classe (oltre che alle funzioni e variabili globali)
  - Non possono usare il puntatore this
  - Non possono esistere versioni static e non-static della stessa funzione
  - NON può essere:
    - Virtuale
    - Dichiarata **const** oppure **volatile**

```
class Myclass {
  static int resource;
public:
  Myclass(){resource = 1;}
    .
    .
  static bool get_resource();
  void free_resource(){ resource = 0; };
};

bool Myclass::get_resource()
{
  if (resource == 0) {
    resource = 1;
   return true;
  } else return false;
}
```

```
main()
{
  Myclass m1, m2;
        .
        .

  if (Myclass::get_resource())
    cout<<" risorsa assegnata ad m1";
        .
        .
        .
    if (m1.free_resource())
    cout<<" m2 non può accedere la risorsa";
}
```

Può essere chiamata in modo indipendente da qualsiasi oggetto

Oppure con la normale sintassi degli oggetti

- Le funzioni static vengono solitamente usate per “preinizializzare” variabili static senza usare istanze della classe:

```
class Myclass {
  static int i;
    .
    .
public:
  Myclass();
  static void init(int x){i = x;}
    .
    .
};
```

```
int Myclass::i // Definisce i


main()
{
  Myclass::init(100); // Inizializza i
  Myclass m1, m2;
        .
        .
}
```

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

```
int main()
{
    .
    .

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

    .
    .
 return 0;
}
```

Le funzioni friend

- Nella definizione di una classe è possibile elencare quali funzioni, esterne alla classe possono accedere ai membri privati della classe. Queste funzioni sono dette *friend.*
- Per dichiarare una funzione friend è necessario includere il prototipo nella classe, facendola precedere dalla parola chiave **friend**:
- Sono particolarmente utili quando due o più classi contengono membri correlati con altre parti del programma.

Le funzioni friend: Esempio

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

```
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;
}
```

```
main()
{
  Myclass m;

 m.set_ab(2, 4);

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

Sum non è membro di Myclass, ma avendola dichiarata friend può accedere ai suoi membri privati

Le 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 C1 {
   .
   .
   .
  friend class C2;
};
```

- Osserviamo che le funzioni membro di C2 possono accedere ai mebri di C1, ma NON VICEVERSA.

Overloading degli operatori

Overloading degli operatori

- La definizione del costruttore di assegnazione è un esempio di ***overloading degli operatori*** che consente di attribuire ulteriori significati agli operatori del linguaggio.

- In C++ è possibile eseguire l'overloading della maggior parte degli operatori, 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 sui tipi standard (int, float, ecc.).

- L'overloading degli operatori consente di integrare le nuove classi create dall'utente nell'ambiente di programmazione.

- L'overloading degli operatori è alla base delle operazioni di I/O del C++.

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

- Una funzione operatore 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.

Funzioni operator membro

- Le funzioni operator membro di una classe hanno la seguente forma generale:

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

Dove # è il simbolo del generico 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 l'operatore è unario la funzione non ha argomenti.

Overloading degli operatori: regole

- È possibile modificare il significato di un operatore esistente, non è possibile creare nuovi operatori
- Non e' 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

Overloading: il compilatore

- Il compilatore, quando incontra un'espressione della forma

  x # y

  verifica nell'ordine:
  - se nella classe X dell'oggetto x vi è una funzione membro della forma operator#(Y), dove Y è la classe dell'oggetto y;
  - se vi è una funzione non membro della forma operator#(X, Y)

- Se una delle due condizioni è verificata provvede ad invocare la funzione individuata, altrimenti segnala un errore.

Funzioni operator membro: esempio

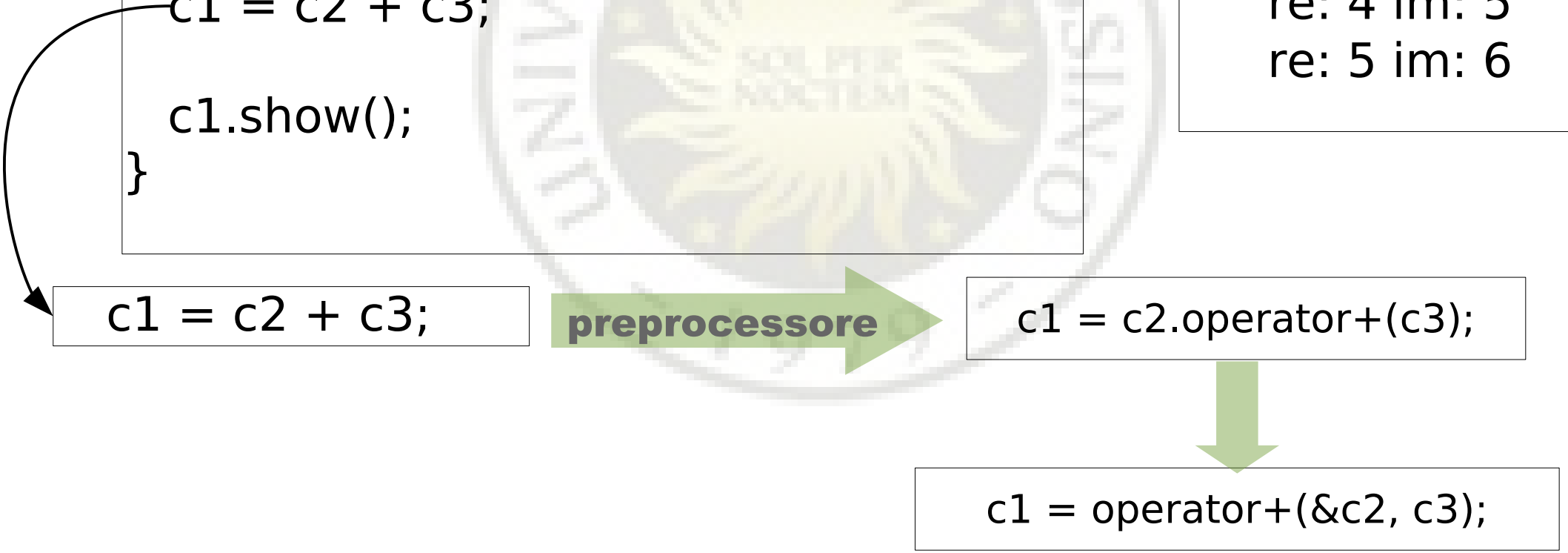

```
class Complex {
    float re;
    float im;
  public:
    Complex(float r=0.0, float i=0.0) {re=r; im=i;}
    float getRe() const {return re; }
    float getIm() const {return im; }
    void setRe(float r) {re=r; }
    void setIm(float i) {im=i; }
    void show();
    Complex operator+(Complex op2);
};
```

```
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;
}
```

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

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

  c1 = c2 + c3;

  c1.show();
}
```

output

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



Funzioni operator: osservazioni

- Quando si esegue l'ovelroading di un operatore binario è l'oggetto di sinistra a generare la chiamata alla funzione operator.
- L'operatore di assegnazione 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.

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

Operatore incremento (prefisso)

- L'operatore prefisso di incremento è un operatore unario e quindi non ha parametri
- Questo operatore prevede la modifica dell'operando:

```
Complex Complex::operator++()
{
  ++re;
  ++im;

  return *this;
}
```

Operatore incremento (postfisso)

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

```
Complex Complex::operator++(int x)
{
  ++re;
  ++im;

  return *this;
}
```

```
main()
{
  Complex c1(1,2), c2(3,5), c3(9,9);

  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

- Viene usato in tutte le espressioni in cui è presente il simbolo = .
- L'assegnazione di default effettua un copia bit a bit. L'overloading è necessario per le classi che hanno un'estensione dinamica.

- Deve essere una funzione membro. 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);
 };
```

```
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;
}
```

Osservazioni

- l'operatore restituisce ***this** ovvero l'oggetto che ha generato la chiamata. Questo accorgimento rende possibile assegnamenti multipli del tipo:

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

- Il passaggio del parametro other per riferimento, ma con il qualificatore **const,** è equivalente al passaggio per valore, ma è più **EFFICIENTE:**

- Si restituisce un reference: evita la chiamata del costruttore di copia

```
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
COSTRUTTORE:0x7ffff81bead0
COSTRUTTORE:0x7ffff81beae0
ASSEGNAZIONE:0x7ffff81bead0
COPIA:0x7ffff81beaf0
ASSEGNAZIONE:0x7ffff81beac0
COPIA:0x7ffff81beab0
```

```
m1
m2
m3
m2
 ?
m1
 ?
```

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

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

restituizione 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));
```

Overloading delle 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 tramite funzioni friend

- L'oveloading può essere eseguito anche per mezzo di funzioni non membro, ma definite friend per la classe in esame.

- In questo caso, il numero di argomenti coincide con il numero di operandi.

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

```
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;
}
```

Operatori membro e funzioni friend: Confronto

- In molti casi, l'overloading di un operatore può essere fatto indifferentemente sia tramite una funzione membro oppure per mezzo di una funzione esterna alla classe dichiarata **friend**.
- In tali casi le funzioni membro sono da preferire
- Ci sono però situazioni in cui le funzioni friend sono preferibili...

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

```
Complex 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!
}
```

- L'operatore visto in precedenza può essere reso più flessibile utilizzando due funzioni esterne 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
}
```

Overloading degli operatori di stream

- Gli operatori << e >> sono stati modificati tramite oveloading in C++ per effettuare l'oveloading delle operazioni di I/O dei tipi standard

- È possibile effettuare l'overloading di questi operatori anche per i nuovi tipi (classi) definiti dall'utente

- L'overloading degli operatori di I/O per una qualsiasi classe può essere fatto solo per mezzo di funzioni (esterne) dichiarate friend

- Perchè?

  Risposta:

  perchè lo stream appare sempre a sinistra nelle operazioni di I/O, e non la variabile da visualizzare

```
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

```
void 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);
   }
```

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

  return;

}
```

```
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 )
    cout<<endl<<”ERRORE: impossibile scrivere l'array!”;

  return;
}
```