# SECONDA PARTE

# (le classi nel linguaggio C++)

# I TIPI CLASSE

# 16.2 Tipi classe (I)

**Nell'esempio precedente, la struttura interna della pila è visibile ai moduli che utilizzano istanze della pila. In questo modo non si riesce a realizzare compiutamente il tipo di dato astratto.**
**Per ovviare a questo problema, il C++ mette a disposizione le classi.**

```
#include <iostream>
using namespace std;

typedef  int T;
const int DIM = 5;
class pila
{     int top;
      T stack[DIM];
public:
       void inip();
       bool empty();
       bool full();
       bool push(T s);
       bool pop(T& s);
       void stampa();
};

int main()
{
      pila st;
      st.inip();
      st.push(1);
      st.top = 10;     // ERRORE!
                       // 'int pila::top' is private within this context
}
```

# 16.2 Tipi classe (II)

*basic-class-type-declaration*
    *class-type-specifier* **;**
*class-type-specifier*
    **class** *identifier*|**opt {** *class-element-seq* **}**
*class-element*
    *access-indicator*|**opt** *class-member-section*
*access-indicator*
    *access-specifier* **:**
*access-specifier*
    **private**
    **protected**
    **public**

**In genere utilizzeremo la forma semplificata seguente:**

**class *nome***
**{** ***parte privata***
**protected:**
    ***parte protetta***
**public:**
    ***parte pubblica***
**};**

**Un membro di una classe può essere:**
**- un tipo (enumerazione o struttura);**
**- un campo dati (oggetto non inizializzato);**
**- una funzione (dichiarazione o definizione);**
**- una classe (diversa da quella della classe a cui appartiene).**

# 16.2 Tipi classe (III)

```
// Numeri Complessi (parte reale, parte immaginaria)
#include<iostream>
using namespace std;
class complesso
{
    double re, im;
public:
    void iniz_compl(double r, double i) {re = r; im = i;}
    double reale() {return re;}
    double immag() {return im;}
    /* ... */
    void scrivi() {cout << '(' << re << ", " << im << ')';}
};

int main()
{
    complesso c1, c2;
    c1.iniz_compl(1.0, -1.0);          //inizializzazione
    c1.scrivi(); cout << endl;         // (1, -1)

    c2.iniz_compl(10.0, -10.0);
    c2.scrivi(); cout << endl;         // (10, -10)

    complesso* cp = &c1;
    cp->scrivi(); cout << endl;        // (1, -1)

    return 0;
}
```

```
(1, -1)
(10, -10)
(1, -1)
```

# 16.2 Tipi classe (IV)

```
// Esempio Numeri Complessi

#include<iostream>
using namespace std;
class complesso
{
public:
    void iniz_compl(double r, double i) {re = r; im = i;}
    double reale() {return re;}
    double immag() {return im;}
    /* ... */
    void scrivi() {cout << '(' << re << ", " << im << ')';}
private:
    double re, im;
};

int main(){
    complesso c1, c2;
    c1.iniz_compl(1.0, -1.0);
    c1.scrivi(); cout << endl;          // (1, -1)

    c2.iniz_compl(10.0, -10.0);
    c2.scrivi(); cout << endl;          // (10, -10)

    complesso* cp = &c1;
    cp->scrivi(); cout << endl;         // (1, -1)
    return 0;
}
```

```
(1, -1)
(10, -10)
(1, -1)
```

# 16.2 Tipi classe (V)

```
// Numeri Complessi (rappresentazione polare)
#include<iostream>
#include<cmath>
using namespace std;
class complesso
{    double mod, arg;
public:
     void iniz_compl(double r, double i)
          {mod = sqrt(r*r + i*i); arg = atan(i/r);}
     double reale() {return mod*cos(arg);}
     double immag() {return mod*sin(arg);}
     /* ... */
     void scrivi()
          {cout << '(' << reale() << ", " << immag() << ')';}
};
int main(){
     complesso c1, c2;
     c1.iniz_compl(1.0, -1.0);
     c1.scrivi(); cout << endl;          // (1, -1)

     c2.iniz_compl(10.0, -10.0);
     c2.scrivi(); cout << endl;          // (10, -10)

     complesso* cp = &c1;
     cp->scrivi(); cout << endl;         // (1, -1)

     return 0;
}
```

```
(1, -1)
(10, -10)
(1, -1)
```

# 16.2 Tipi classe (VI)

**Le funzioni membro definite nella dichiarazione di una classe sono funzioni inline.**
**Le funzioni membro possono anche essere definite esternamente utilizzando l'operatore di risoluzione di visibilita'.**

```
#include<iostream>
using namespace std;
class complesso
{
public:
    void iniz_compl(double r, double i);
    double reale();
    double immag();
    /* … */
    void scrivi();
private:
    double re, im;
};

void complesso:: iniz_compl(double r, double i)
{re = r; im = i;}

double complesso:: reale()
{return re;}

double complesso::immag()
{return im;}

void complesso::scrivi()
{cout << '(' << re << ", " << im << ')';}
```

# 16.3 Operazioni su oggetti classe

**Un oggetto appartenente ad una classe si chiama oggetto classe o istanza della classe**

```
class complesso
{/*…*/};

int main()
{
    complesso c1;
    c1.iniz_compl(1.0, -1.0);
    complesso c2 = c1, c3(c2);
        //Inizializzazione - ricopiatura membro a membro
    c1.scrivi(); cout << endl;          // (1, -1)
    c2.scrivi(); cout << endl;          // (1, -1)
    c3.scrivi(); cout << endl;          // (1, -1)

    complesso *pc1 = new complesso(c1);
    pc1->scrivi(); cout << endl;        // (1,-1)

    complesso* pc2 = &c1;
    pc2->scrivi(); cout << endl;        // (1, -1)


    return 0;
}
```

```
(1, -1)
(1, -1)
(1, -1)
(1, -1)
(1, -1)
```

# 16.3 Operazioni su oggetti classe

```
class complesso
{/*…*/};

complesso somma(complesso a, complesso b)
{
    complesso s;
    s.iniz_compl(a.reale()+b.reale(),
                 a.immag() + b.immag());
    return s;
}

int main()
{
    complesso c1, c2, c3;
    c1.iniz_compl(1.0, -1.0);
    c2 = c1; //Assegnamento (ricopiatura membro a membro)
    c1.scrivi(); cout << endl;          // (1, -1)
    c2.scrivi(); cout << endl;          // (1, -1)

    c3 = somma(c1,c2);        // oggetti argomento di funzioni
                              // e restituiti da funzioni
    c3.scrivi(); cout << endl;          // (2, -2)

    return 0;
}
```

```
(1, -1)
(1, -1)
(2, -2)
```

**ATTENZIONE: non esistono altre operazioni predefinite**

# 16.4 Puntatore this (I)

**Nella definizione di una funzione membro, il generico oggetto a cui la funzione viene applicata può essere riferito tramite il puntatore costante predefinito this**

```
class complesso
{/*…*/
    complesso scala(double s)  // restituisce un valore
    {
        re *= s;
        im *= s;
        return *this;
    }
};

int main()
{
    complesso c1;
    c1.iniz_compl(1.0, -1.0);
    c1.scala(2);
    c1.scrivi(); cout << endl;          // (2, -2)

    complesso c2;
    c2.iniz_compl(1.0, -1.0);
    c2.scala(2).scala(2);
    c2.scrivi(); cout << endl;          // (2, -2)

    return 0;
}
```

```
(2, -2)
(2, -2)
```

# 16.4 Puntatore this (II)

```
class complesso
{/*…*/
    complesso& scala(double s)   // restituisce un riferimento
    {
        re *= s;
        im *= s;
        return *this;
    }
};

int main()
{
    complesso c1;
    c1.iniz_compl(1.0, -1.0);
    c1.scala(2);
    c1.scrivi(); cout << endl;          // (2, -2)

    complesso c2;
    c2.iniz_compl(1.0, -1.0);
    c2.scala(2).scala(2);
    c2.scrivi(); cout << endl;          // (4, -4)

    return 0;
}
```

```
(2, -2)
(4, -4)
```

# 16.5 Visibilità a livello di classe (I)

**Una classe individua un *campo di visibilità*.**

- **Gli identificatori dichiarati all'interno di una classe sono visibili dal punto della loro dichiarazione fino alla fine della classe stessa.**
- **Nel corpo delle funzioni membro sono visibili tutti gli identificatori presenti nella classe (anche quelli non ancora dichiarati).**
- **Se nella classe viene riutilizzato un identificatore dichiarato all'esterno della classe, la dichiarazione fatta nella classe nasconde quella più esterna.**
- **All'esterno della classe possono essere resi visibili mediante l'operatore di risoluzione di visibilità :: applicato al nome della classe:**
  - **Le funzioni membro, quando vengono definite;**
  - **Un tipo o un enumeratore, se dichiarati nella parte pubblica della classe**
  - **Membri statici**
- **L'operatore di visibilità non può essere utilizzato per i campi dati non statici.**

# 16.5 Visibilità a livello di classe (II)

```
class grafica
{    // …
     public:
          enum colore { rosso, verde, blu };
};

class traffico
{    // …
public:
     enum colore { rosso, giallo, verde };
     colore stato();
// …
};

traffico::colore traffico::stato()
{  /* … */  }


// …
grafica::colore punto;
traffico::colore semaforo;

int main()
{
     // …
     punto = grafica::rosso;
     semaforo = traffico::rosso;
     //…
}
```

# 16.5 Visibilità a livello di classe (III)

**Nella dichiarazione di una classe si può dichiarare un'altra classe (classe annidata). In questo caso, per riferire membri della classe annidata esternamente alla classe che la contiene devono essere usati due operatori di risoluzione di visibilità.**
**Nessuna delle due classi ha diritti di accesso alla parte privata dell'altra.**

```
class A
{
    class B
    {
        int x;
    public:
        void iniz_B(int);
    };
    int y;
public:
    void iniz_A();
};

void A::iniz_A(){ y = 0; }
void A::B::iniz_B(int n) {x = n;}

// void A::iniz_A(){x = 3; }  ERRORE: A non può accedere
                              // alla parte privata di B

// void A::B::iniz_B(int n){y = n;} // ERRORE: B non puo'
                                    // accedere alla parte privata
                                    // di A
```

# 16.6 Modularità e ricompilazione (I)

**L'uso delle classi permette di scrivere programmi in cui l'interazione tra moduli è limitata alle interfacce.**

**Quando si modifica la parte dati di una classe, anche se gli altri moduli utilizzano solo l'interfaccia, si rende necessaria la ricompilazione dei moduli. Infatti, se in un modulo cliente viene definito un oggetto classe, il compilatore ha bisogno di sapere la dimensione dell'oggetto.**

**Ai fini della ricompilazione non è appropriato separare una classe in interfaccia e struttura interna, ma piuttosto effettuare una suddivisione in file.**

**Un file di intestazione che contiene la dichiarazione della classe e deve essere incluso in tutti i moduli cliente.**

**Un file di realizzazione che contiene tutte le definizioni delle funzioni membro.**

**La modifica di un file di intestazione richiede la ricompilazione di tutti i file che lo includono.**

**La modifica di un file di realizzazione richiede solo la ricompilazione di questo file.**

# 16.6 Modularità e ricompilazione (II)

```
// file complesso.h
class complesso
{
    double re, im;
public:
    void iniz_compl(double r, double i);
    double reale();
    double immag();
    /* ... */
    void scrivi();
    complesso& scala(double s);
};

// file complesso.cpp

#include<iostream>
#include "complesso.h"
using namespace std;

void complesso:: iniz_compl(double r, double i)
{re = r; im = i;}

double complesso:: reale() {return re;}

double complesso::immag() {return im;}

void complesso::scrivi()
    {cout << '(' << re << ", " << im << ')';}

complesso& complesso::scala(double s)
{       re *= s;   im *= s;       return *this;}
```

# 16.6 Modularità e ricompilazione (III)

```
// file main.cpp

#include<iostream>
#include "complesso.h"
using namespace std;
int main()
{
    complesso c1;
    c1.iniz_compl(1.0, -1.0);
    c1.scala(2);
    c1.scrivi(); cout << endl;          // (2, -2)

    complesso c2;
    c2.iniz_compl(1.0, -1.0);
    c2.scala(2).scala(2);
    c2.scrivi(); cout << endl;          // (4, -4)

    return 0;
}
```

# 16.7 Funzione globali

**Funzioni globali: funzioni che non sono membro di alcuna classe.**
**Non possono accedere alla parte privata delle classi e non possono usare il puntatore this.**

```
#include<iostream>
#include "complesso.h"

complesso somma(const complesso& a, const
                              complesso& b)
{
     complesso s;
     s.iniz_compl(a.reale()+b.reale(),
                         a.immag() + b.immag());
     return s;
}

using namespace std;
int main()
{
     complesso c1, c2, c3;
     c1.iniz_compl(1.0, -1.0);
     c2.iniz_compl(2.0, -2.0);

     c3 = somma(c1,c2);
     c3.scrivi(); cout << endl;          // (3, -3)

     return 0;
}
```

```
(3, -3)
```

# 16.8 Costruttori (I)

**Costruttore: funzione membro il cui nome è il nome della classe. Se definita viene invocata automaticamente tutte le volte che viene creata un'istanza di classe (subito dopo che è stata riservata la memoria per i campi dati)**

```
// file complesso.h
class complesso
{
    double re, im;
public:
    complesso(double r, double i);
    double reale();
    double immag();
    /* ... */
    void scrivi();
};

// file complesso.cpp
/*….*/
complesso:: complesso(double r, double i)
{re = r; im = i;}

// file main.cpp
/*….*/
int main()
{
   complesso c1(1.0, -1.0);
   c1.scrivi(); cout << endl;              // (1, -1)

   complesso c2;      // ERRORE: non esiste complesso()
   complesso c3(3);  // ERRORE: non esiste complesso(int)
}
```

# 16.8 Costruttori default(I)

**Per definire oggetti senza inizializzatori occorre definire un costruttore di default che può essere chiamato senza argomenti. Due meccanismi.**

**1) Meccanismo dell'overloading**

```
// file complesso.h
class complesso
{
    double re, im;
public:
    complesso();
    complesso(double r, double i);
    double reale();
    double immag();
    /* ... */
    void scrivi();
};
```

```
// file complesso.cpp
/*….*/
complesso :: complesso()   // costruttore di default
{re = 0; im = 0;}
complesso :: complesso(double r, double i)
{re = r; im = i;}
```

```
// file main.cpp
/*….*/
int main()
{   complesso c1(1.0, -1.0);
    c1.scrivi(); cout << endl;          // (1, -1)
    complesso c2;
    complesso c3(3.0);                  // ERRATO
}
```

# 16.8 Costruttori default(II)

**2) Meccanismo degli argomenti default**

```
// file complesso.h
class complesso
{
    double re, im;
public:
    complesso(double r = 0, double i = 0);
    double reale();
    double immag();
    /* … */
    void scrivi();
};
```

```
// file complesso.cpp
/*….*/
complesso :: complesso(double r, double i)
{re = r; im = i;}
```

```
// file main.cpp
/*….*/
int main()
{
    complesso c1(1.0, -1.0);      // (1, -1)
    complesso c2;                 // (0, 0)
    complesso c3(3.0);            // (3, 0)
    complesso c4 = 3.0;           // (3, 0)
    // possibile quando il costruttore può essere
    // invocato con un solo argomento
}
```

**ATTENZIONE: I due meccanismi non possono essere utilizzati contemporaneamente.**

# 16.8 Costruttori per allocare memoria

**Spesso un costruttore richiede l'allocazione di memoria libera per alcuni membri dell'oggetto da costruire**

```
// file stringa.h
class stringa
{
       char* str;
public:
      stringa(const char s[]);
// ...
};

// file stringa.cpp
stringa::stringa(const char s[])
{
     str = new char[strlen(s)+1];
     strcpy(str, s);     // copia la stringa s nella stringa str
}
// ...

// file main.cpp
int main()
{
     stringa inf("Fondamenti di Progr.");
      /*...*/
}
```

**Quando definiamo una variabile di tipo stringa, viene riservata memoria per *str* e quindi viene richiamato il costruttore che alloca un array della dimensione richiesta.**

# 16.8 Costruttori per oggetti dinamici

**I costruttori definiti per una classe vengono chiamati implicitamente anche quando un oggetto viene allocato dinamicamente**

```
// file main.cpp

#include<iostream>
#include "complesso.h"
int main()
{
    complesso* pc1 = new complesso(3.0, 4.0);
    complesso* pc2 = new complesso(3.0);
    complesso* pc3 = new complesso;
    /*…*/
}
```

# 16.9 Distruttori

**Distruttore: funzione membro che viene invocata automaticamente quando un oggetto termina il suo tempo di vita.**
**Un distruttore non ha argomenti.**
**Obbligatori quando il costruttore alloca memoria dinamica.**

```
// file stringa.h
class stringa
{
      char* str;
public:
     stringa(const char s[]);
     ~stringa();                    // distruttore
// ...
};

// file stringa.cpp
stringa::stringa(const char s[])
{    str = new char[strlen(s)+1];
     strcpy(str, s);     // copia la stringa s nella stringa str
}
stringa::~stringa()
{    delete[] str;}
/*…*/

// file main.cpp
/*…*/
int main()
{    stringa* ps = new stringa("Fondamenti di Progr.");
      /*...*/
     delete ps;
}
```

# 16.9 Regole di chiamata

**I costruttori vengono richiamati con le regole seguenti:**

1. **Per gli oggetti statici, all'inizio del programma;**
2. **Per gli oggetti automatici, quando viene incontrata la definizione;**
3. **Per gli oggetti dinamici, quando viene incontrato l'operatore new;**
4. **Per gli oggetti membri di altri oggetti, quando questi ultimi vengono costruiti.**

**I distruttori vengono richiamati con le regole seguenti:**

1. **Per gli oggetti statici, al termine del programma;**
2. **Per gli oggetti automatici, all'uscita dal blocco in cui sono definiti;**
3. **Per gli oggetti dinamici, quando viene incontrato l'operatore delete;**
4. **Per gli oggetti membri di altri oggetti, quando questi ultimi vengono distrutti.**

**Gli oggetti con lo stesso tempo di vita vengono distrutti nell'ordine inverso a quello in cui sono definiti.**

# 16.9 Costruttori/Distruttori Esempio

```
// file stringa.cpp
/*…*/
stringa::stringa(const char s[])
{    str = new char[strlen(s)+1];
     strcpy(str, s);     // copia la stringa s nella stringa str
     cout << "C " << str << endl;
}

stringa::~stringa()
{    cout << "D " << str << endl;
     delete[] str;
}

// file main.cpp
#include <cstdlib>
#include "stringa.h"
int main()
{    stringa* ps = new stringa("Fondamenti di Progr.");
     {
          stringa s("Reti Logiche");
      }
     delete ps;
     /*...*/
     return 0;
}
```

```
C Fondamenti di Progr.
C Reti Logiche
D Reti Logiche
D Fondamenti di Progr.
```

# 16.10 Costruttori di copia (I)

**Costruttore di copia predefinito: costruttore che agisce fra due oggetti della stessa classe effettuando una ricopiatura membro a membro dei campi dati.**

**Viene applicato:**

1. **Quando un oggetto classe viene inizializzato con un altro oggetto della stessa classe**
2. **Quando un oggetto classe viene passato ad una funzione come argomento valore;**
3. **Quando una funzione restituisce come valore un oggetto classe (mediante l'istruzione return)**

**Nel caso della classe complesso il costruttore di copia predefinito ha la seguente forma:**
**complesso(const complesso& c) { re = c.re; im = c.im;};**

# 16.10 Costruttori di copia (II)

```
// file main.cpp
#include "complesso.h"
complesso fun (complesso c)
{
  /*….*/
  return c;
}

int main()
{
     complesso c1 (3.2, 4);
     c1.scrivi(); cout << endl;          // <3.2, 4>

     complesso c2 (c1);                  // costruttore di copia
     c2.scrivi(); cout << endl;          // <3.2, 4>

     complesso c3 = c1;                   // costruttore di copia
     c3.scrivi(); cout << endl;          // <3.2, 4>

     c3 = fun(c1);        // istanza di funzione
                          // costruttore di copia applicato 2 volte:
                          // chiamata di funzione
                          // restituzione del valore
     c3.scrivi(); cout << endl;          // <3.2, 4>

     return 0;
}
```

# 16.10 Costruttori di copia (III)

**Per osservare quando il costruttore di copia viene effettivamente richiamato, ridefiniamo il costruttore di copia nella classe complesso ed eseguiamo lo stesso programma principale**

```
// file complesso.h
class complesso
{
    double re, im;
public:
    complesso(const complesso&);   // costruttore di copia
    /* ... */
};
```

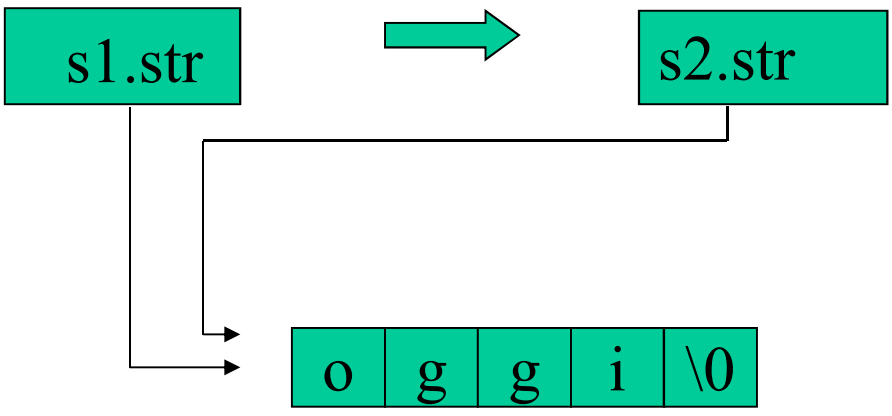
```
// file complesso.cpp
/*….*/
complesso :: complesso(const complesso& c)
{       re = c.re; im = c.im;
        cout << "Costruttore di copia " << endl;
}
```

```
(3.2, 4)
Costruttore di copia
(3.2, 4)
Costruttore di copia
(3.2, 4)
Costruttore di copia
Costruttore di copia
(3.2, 4)
```

# 16.10 Costruttori di copia (IV)

**Il costruttore di copia deve essere ridefinito per quelle classi che utilizzano memoria libera e prevedono un distruttore.**

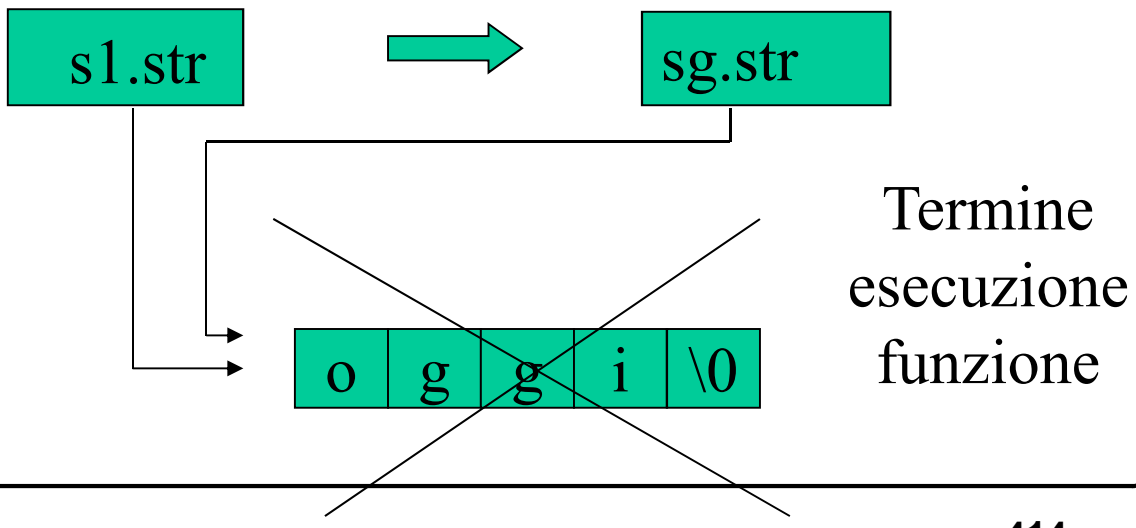
```
// file stringa.h
class stringa
{
      char* str;
public:
     stringa(const char s[]);
     ~stringa();                    // distruttore
// ...
};

// file main.cpp
#include <cstdlib>
#include "stringa.h"
int main()
{
     stringa s1("oggi");
     stringa s2(s1);          // costruttore di copia predefinito

     return 0;
}
```



# 16.10 Costruttori di copia (V)

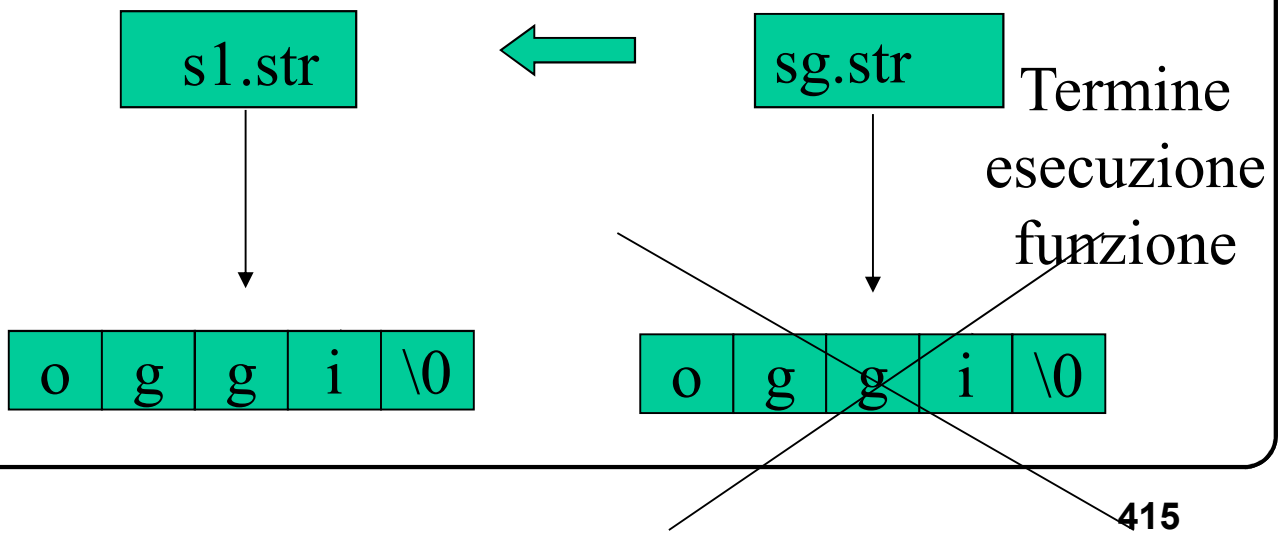
**Consideriamo l'esempio seguente.**

```
// file main.cpp
#include <cstdlib>
#include "stringa.h"

void ff(stringa sg)
{
    // …
}

int main()
{
    stringa s1("oggi");
    ff(s1);

    return 0;
}
```



# 16.10 Costruttori di copia (VI)

```
// file stringa.h
class stringa
{
        char* str;
public:
       stringa(const stringa &);
// …
};

// file stringa.cpp
/*…*/
stringa::stringa(const stringa& s)
{     str = new char[strlen(s.str) + 1];
      strcpy(str, s.str);
}

// file main.cpp
/*…*/
int main()
{
      stringa s1("oggi");
      ff(s1);
      /*…*/
}
```



# 16.10 Costruttori di copia (VI)

**Se il costruttore di copia non viene ridefinito, viene usato il costruttore di copia prefefinito.**

**Per impedirne l'utilizzo, occorre inserire la sua dichiarazione nella parte privata della classe stessa senza alcuna ridefinizione.**

**Nel caso in cui il costruttore di copia venga nascosto, non si possono avere funzioni che abbiano argomenti valore del tipo della classe o un risultato valore del tipo della classe.**

```
// file stringa.h
class stringa
{
    char* str;
    stringa(const stringa &);
public:
     // ...
};

// file main.cpp
/*…*/
void ff(stringa sg)              // ERRORE
{   // …
}

void ff(stringa& sg)             // OK
{
    // …
}
```

# 16.11 Funzioni friend (I)

**Una funzione è *friend* (*amica*) di una classe se una sua dichiarazione, preceduta dalla parola chiave friend, appare nella dichiarazione di tale classe. Una funzione friend può accedere ai membri pubblici e privati della classe, usando i selettori di membro.**

```
// file complesso.h
class complesso
{     double re, im;
public:
 double reale();
 double immag();
 /* ... */
};

// file main.cpp
complesso somma(const complesso& a, const
                          complesso& b)
{
 complesso s(a.reale()+b.reale(), a.immag() + b.immag());
  return s;                    // Non si puo' accedere ai
}                              // membri privati della classe

int main()
{
 complesso c1 (3.2, 4);
 complesso c2 (c1);
 complesso c3 = somma(c1,c2);
 c3.scrivi(); cout << endl;              // (6.4, 8)

 return 0;
}
```

# 16.11 Funzioni friend (II)

```
// file complesso.h
class complesso
{     double re, im;
public:
 double reale();
 double immag();
 friend complesso somma(const complesso& a,
                        const complesso& b);
 /* ... */
};

// file main.cpp
complesso somma(const complesso& a, const
                complesso& b)
{
 complesso s(a.re+b.re, a.im + b.im);
  return s;                  // Si puo' accedere ai
}                            // membri privati della classe

int main()
{
 complesso c1 (3.2, 4);
 complesso c2 (c1);
 complesso c3 = somma(c1,c2);
 c3.scrivi(); cout << endl;          // (6.4, 8)

 return 0;
}
```

# 16.11 Funzioni friend (III)

```
// Esempio di utilizzo della FRIEND nel caso di una classe
// che utilizza un'altra classe:
//      CASO A) classe utilizzatrice FRIEND (per intero) della classe utilizzata

#include<iostream>
using namespace std;

class complesso{            // dichiarazione della classe utilizzata
    double re, im;
public:
   complesso(double r = 0, double i = 0){re=r; im=i;}
   /* seguono le altre funzioni membro della classe complesso... */
   friend class vettoreComplesso; // FRIEND tutta la classe utilizzatrice
};

class vettoreComplesso // dichiarazione della classe utilizzatrice
{  complesso *vett;    int size;
public:
   vettoreComplesso(int sz){
      ( sz <= 0 ) ? size = 10 : size = sz ;
      vett = new complesso[size]; // chiamata al costruttore di default
                                          // su ogni complesso del vettore
   }
   double moduloQuadro(){
      double mQ = 0;
      for (int i=0; i<size; i++){
         // essendo friend vettoreComplesso puo' accedere a .re e .im
         // senza dover chiamare le funzioni di accesso reale() e immag()
         mQ += vett[i].re*vett[i].re + vett[i].im*vett[i].im;
      }
      return mQ;
   }
}; // fine classe vettorecomplesso

int main(){
      vettoreComplesso v(5);
      cout<<"Modulo quadro di v: ";
      cout<<v.moduloQuadro()<<endl;
      return 0;
}
```

# 16.11 Funzioni friend (IV)

```
// Esempio di utilizzo della FRIEND nel caso di una classe
// che utilizza un'altra classe:
//          CASO B)  solo alcuni metodi della classe utilizzatrice
//                   sono FRIEND di quella utilizzata

#include<iostream>
using namespace std;

class complesso;            // dichiarazione incompleta della classe utilizzata
class vettoreComplesso  // dichiarazione (completa) della classe utilizzatrice
{
   complesso *vett;      int size;
public:
   vettoreComplesso(int);
   double moduloQuadro();
};

class complesso{ // dichiarazione (completa) della classe utilizzata
  double re,  im;
public:
 complesso(double r = 0, double i = 0){re=r; im=i;}
 friend double vettoreComplesso::moduloQuadro();  // FRIEND un metodo solo
};

// definizione delle funzioni membro della classe utilizzatrice
vettoreComplesso::vettoreComplesso(int sz){
     size = sz;    vett = new complesso[size]; // chiamata al costruttore di default
}

double vettoreComplesso::moduloQuadro(){
   double mQ = 0;
   for (int i=0; i<size; i++){
      mQ += vett[i].re*vett[i].re + vett[i].im*vett[i].im;
   }
   return mQ;
}

int main(){
   vettoreComplesso v(5);    cout<<"Modulo quadro di v: ";
   cout<<v.moduloQuadro()<<endl;
   return 0;
}
```

# 17.1 Overloading di operatori (I)

- Tipo di dato astratto: può richiedere che vengano definite delle operazioni tipo somma, prodotto, etc.
- Opportuno utilizzare gli stessi operatori usati per le operazioni analoghe sui tipi fondamentali.
- Necessaria la ridefinizione degli operatori
- La ridefinizione di un operatore ha la forma di una definizione di funzione, il cui identificatore è costituito dalla parola chiave *operator* seguita dall'operatore che si vuole ridefinire, e ogni occorrenza dell'operatore equivale ad una chiamata alla funzione.
- Un operatore può designare una funzione membro o una funzione globale.

**Se Θ è un operatore unario**

1. Funzione membro *operatorΘ()*
2. Funzione globale *operatorΘ(x)*

**Se Θ è un operatore binario**

1. Funzione membro *operatorΘ(x)*
2. Funzione globale *operatorΘ(x,y)*

**ATTENZIONE: si possono usare solo operatori già definiti nel linguaggio e non se ne possono cambiare le proprietà**

**ATTENZIONE: una nuova definizione di un operatore deve avere fra gli argomenti almeno un argomento di un tipo definito dall'utente**

# 17.1 Overloading di operatori (II)

**Ridefinizione dell'operatore somma per i numeri complessi come funzione membro**

```
//file complesso.h
class complesso
{     double re, im;
public:
      complesso(double r = 0, double i = 0);
      complesso operator+(const complesso& );
 /*…*/
};

// file complesso.cpp
/*…*/
complesso complesso::operator+(const complesso& x)
{
        complesso z(re+x.re,im+x.im);
        return z;
}

// file main.cpp

#include<iostream>
#include "complesso.h"
int main()
{
      complesso c1 (3.2, 4);
      complesso c2 (c1);
      complesso c3 = c1 + c2;
      c3.scrivi(); cout << endl;              // (6.4, 8)
      return 0;
}
```

# 17.1 Overloading di operatori (III)

**Ridefinizione dell'operatore somma per i numeri complessi come funzione globale**

```
//file complesso.h
class complesso
{     double re, im;
public:
      complesso(double r = 0, double i = 0);
      friend complesso operator+(const complesso&,
                                 const complesso&);
 /*…*/
};
// file complesso.cpp
/*…*/
complesso operator+(const complesso& x,const
                                 complesso& y)
{
        complesso z(x.re+y.re, x.im+y.im);
        return z;
}
// file main.cpp

#include<iostream>
#include "complesso.h"
int main()
{
      complesso c1 (3.2, 4);
      complesso c2 (c1);
      complesso c3 = c1 + c2;
      c3.scrivi(); cout << endl;              // (6.4, 8)
      return 0;
}
```

# 17.1 Overloading di operatori (IV)

**Operatori incremento**

```
//file complesso.h
class complesso
{     double re, im;
public:
      complesso& operator++();
      complesso operator++(int);
      //argomento fittizio che individua l'operatore postfisso
      /*…*/
};

// file complesso.cpp
/*…*/
complesso& complesso::operator++()
{     re++;  im++;
      return *this;
}
complesso complesso::operator++(int)
{
      complesso app = *this;
      re++;  im++;
      return app;
}
```

# 17.1 Overloading di operatori (V)

**Operatori incremento**

```
// file main.cpp
#include<iostream>
#include "complesso.h"
int main()
{
        complesso c1 (3.2, 4);
        complesso c2 (c1);
        complesso c3 = c1 + c2;
        c3.scrivi(); cout << endl;              // (6.4, 8)
        c1++;
        c1.scrivi(); cout << endl;              // (4.2, 5)
        ++c2;
        c2.scrivi(); cout << endl;              // (4.2, 5)
        c1++++++;
        c1.scrivi(); cout << endl;              // (5.2, 6)
        ++++++c2;
        c2.scrivi(); cout << endl;              // (7.2, 8)

        return 0;
}
```

# 17.1 Overloading di operatori (VI)

**Tutti gli operatori di assegnamento si ridefiniscono come funzioni membro e restituiscono un riferimento all'oggetto**

**Nella scelta fra le diverse versioni di una funzione sovrapposta (o di un operatore sovrapposto), il compilatore tiene conto delle possibili conversioni di tipo: queste possono essere conversioni fra tipi fondamentali (implicite o indicate dal programmatore), o conversioni definite dal programmatore che coinvolgono tipi classe.**

**Non ci sono priorità fra le possibili conversioni. Se esistono diverse possibilità, il compilatore segnala questa ambiguità.**

**ATTENZIONE: non assegnate ad un operatore un significato non intuitivo**

**ATTENZIONE: le versioni predefinite degli operatori assicurano alcune equivalenze. Per esempio, ++x è equivalente a x+=1. Quando si ridefinisce un operatore, le equivalenze tra gli operatori non valgono automaticamente. Sarà compito del programmatore fare in modo che tali equivalenze vengano rispettate.**

# 17.1 Overloading di operatori (VII)

**Ambiguità non risolta dal compilatore.**

```
//file complesso.h
class complesso
{     double re, im;
public:
      complesso(double r = 0, double i = 0);
      complesso operator+(const complesso& );
      operator double(){ return re+im;}
 /*…*/
};

// file complesso.cpp
/*…*/
complesso complesso::operator+(const complesso& x)
{     complesso z(re+x.re,im+x.im);
      return z;
}
// file main.cpp

#include<iostream>
#include "complesso.h"
int main()
{     complesso c1 (3.2, 4), c2;
      cout << double(c1) << endl;     // 7.2
      c2 = c1 + 3.0;
// ERRORE: ambiguous overload for 'operator+' in 'c1 +
// 3.0e+0'  candidates are:
// operator+(double, double) <built-in>
// complesso complesso::operator+(const complesso&)

      return 0;
}
```

# 17.2 Simmetria tra gli operatori (I)

**Gli operatori ridefiniti come funzioni membro non sono simmetrici.**

```
//file complesso.h
class complesso
{     double re, im;
public:
      complesso(double r = 0, double i = 0);
      complesso operator+(const complesso& );
 /*…*/
};

// file complesso.cpp
/*…*/
complesso complesso::operator+(const complesso& x)
{
        complesso z(re+x.re,im+x.im);
        return z;
}


// file main.cpp
#include<iostream>
#include "complesso.h"
int main()
{
      complesso c1 (3.2, 4), c2;
      c2 = c1 + 3.0;
 // chiamata implicita al costruttore prefefinito  con
 // argomento attuale 3. Somma tra numeri complessi
      c2 = 3.0 + c1;          // ERRORE

      return 0;
}
```

# 17.2 Simmetria tra gli operatori (II)

**Per mantenere la simmetria degli operatori (il primo operando possa essere qualsivoglia e non l'oggetto a cui si applica l'operatore) si devono utilizzare funzioni globali.**

```
//file complesso.h
class complesso
{     double re, im;
public:
      complesso(double r = 0, double i = 0);
      friend complesso operator+(const complesso& x,
                        const complesso& y);
      friend complesso operator+(const complesso& x,
                        double d);
      friend complesso operator+(double d, const
                        complesso& y);
 /*…*/
};

// file complesso.cpp
/*…*/
complesso operator+(const complesso& x,
                  const complesso& y)
{
     complesso z(x.re+y.re,x.im+y.im);
      return z;
}

complesso operator+(const complesso& x,double d)
{
       complesso z(x.re+d,x.im+d);
      return z;
}
```

# 17.2 Simmetria tra gli operatori (III)

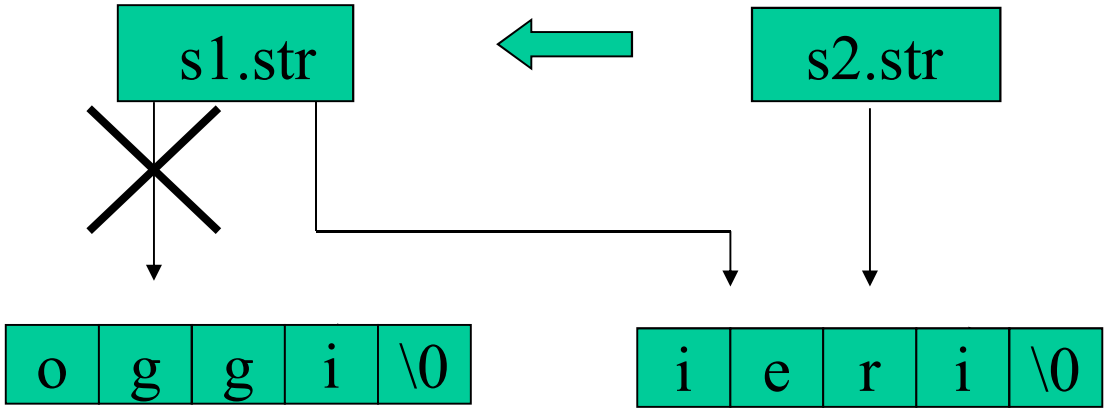
```
// file complesso.cpp
complesso operator+(double d, const complesso& x)
{
      complesso z(x.re+d,x.im+d);
      return z;
}

// file main.cpp

#include<iostream>
#include "complesso.h"
int main()
{
      complesso c1 (3.2, 4);
      complesso c2 = c1 + 4.0;
      c2.scrivi(); cout << endl;          // (7.2, 8)
      c2 = 3.0 + c1;
      c2.scrivi(); cout << endl;          // (6.2, 7)
      c2 = 4.0 + 3.0;                     // Costruttore prefefinito
      c2.scrivi(); cout << endl;          // (7, 0)

      return 0;
}
```

# 17.3 Operatore di assegnamento (I)

**Operatore di assegnamento: predefinito nel linguaggio. Esegue una copia membro a membro. Deve essere ridefinito in presenza di allocazione dinamica della memoria.**

```
// file stringa.h
class stringa
{  char* str;
public:
        stringa(const char s[]);
        ~stringa();                          // distruttore
        stringa(const stringa &);    // costruttore di copia
// ...
};
```

```
// file main.cpp
void fun()
{
        stringa s1("oggi");
        stringa s2("ieri");
        s1 = s2;                          // operatore di assegnamento
  //…
}
// Al termine della funzione viene richiamato il distruttore due
// volte. Comportamento dell'operatore delete quando
// applicato ad un'area gia' liberata non definito!!!!
```



# 17.3 Operatore di assegnamento (II)

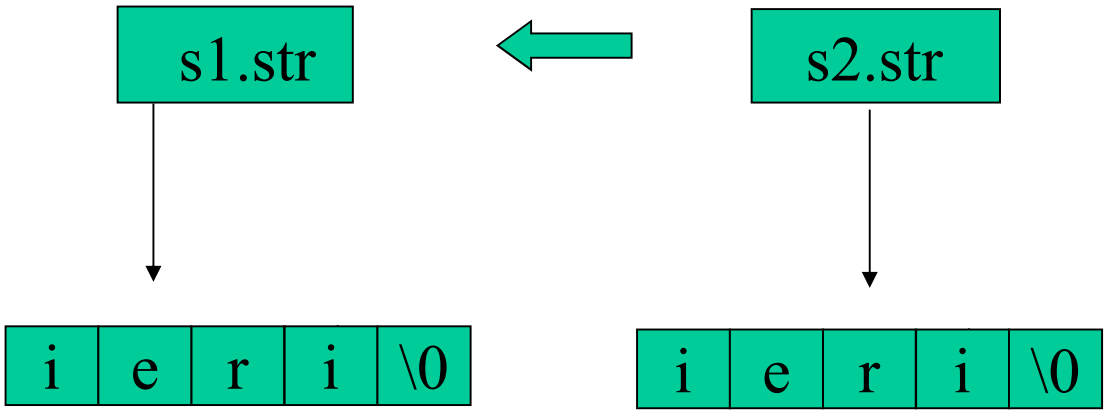
**Operatore di assegnamento deve:**

1. **deallocare la memoria dinamica dell'operando a sinistra**
2. **allocare la memoria della dimensione uguale all'operando destro**
3. **copiare i membri dato e gli elementi dello heap**

**PROBLEMA DI ALIASING: argomento implicito uguale all'argomento esplicito**
**s1 = s1**

```
// file stringa.h
class stringa
{  char* str;
public:
     stringa(const char s[]);
     ~stringa();                        // distruttore
     stringa(const stringa &);  // costruttore di copia
     stringa& operator=(const stringa&); // oper. ass
// ...
};
// file stringa.cpp  NB: l'oggetto di sinistra (sx) è l'oggetto (*this)
stringa& stringa::operator=(const stringa& dx)
{   if (this != &dx)       // CONTROLLO ALIASING
     {
          delete[] str;
          str = new char[strlen(dx.str)+1];
          strcpy(str, dx.str);
     }
     return *this;
}
// NB: il test poteva anche essere scritto: if ( &(*this) != &dx ) ...
```

# 17.3 Operatore di assegnamento (III)




**OTTIMIZZAZIONE:**
**Se la memoria dinamica ha la giusta dimensione possiamo evitare di deallocare e riallocare la memoria**

```
// file stringa.cpp
stringa& stringa::operator=(const stringa& dx)
{   if (this != &dx)
    {
        if (strlen(str) != strlen(dx.str))
        {
            delete[] str;
            str = new char[strlen(dx.str)+1];
        }
        strcpy(str, dx.str);
    }
    return *this;
}
```

# 17.3 Operatori che si possono ridefinire (I)

**Si possono ridefinire tutti gli operatori tranne:**

- **L'operatore risoluzione di visibilità (::)**
- **L'operatore selezione di membro (.)**
- **L'operatore selezione di membro attraverso un puntatore a membro (.*)**

**ATTENZIONE: gli operatori di assegnamento ('='), di indicizzazione ('[]'), di chiamata di funzione ('()') e di selezione di membro tramite un puntatore ('->*') devono essere ridefiniti sempre come funzioni membro.**

**ATTENZIONE: oltre all'operatore di assegnamento, sono predefiniti anche quello di indirizzo ('&') e di sequenza (',').**

# 18.1 Costanti e riferimenti nelle classi (I)

**Il campo dati di una classe può avere l'attributo const: in questo caso il campo deve essere inizializzato nel momento in cui viene definito un oggetto appartenente alla classe stessa**

**INIZIALIZZAZIONE: avviene tramite la lista di inizializzazione del costruttore.**

```
// file complesso.h
class complesso
{     const double re;
      double im;
public:
 complesso(double r = 0, double i = 0);
 complesso operator+(const complesso&);
 /*…*/
};

// file complesso.cpp
complesso:: complesso(double r, double i) : re(r)
{ im = i;}
// anche permessa la scrittura seguente
// complesso:: complesso(double r, double i) : re(r), im(i) {}

complesso :: complesso(const complesso& c) : re(c.re)
{ im = c.im;}
```

**ATTENZIONE: in una classe si possono dichiarare riferimenti non inizializzati, purché siano inizializzati con la lista di inizializzazione**

# 18.2 Membro classe all'interno di classi (I)

**In una classe (classe principale) possono essere presenti membri di tipo classe (classe secondaria) diversa dalla classe principale.**

```
// file record.h
class record                    // classe principale
{
   stringa nome, cognome;
public:
   //….
};
```

**Quando viene dichiarato un oggetto appartenente alla classe principale:**

1. **vengono richiamati i costruttori delle classi secondarie, se definiti, nell'ordine in cui queste compaiono nella dichiarazione dei membri della classe principale;**
2. **quindi, viene eseguito il corpo del costruttore della classe principale, se definito.**

**Quando un oggetto appartenente alla classe principale viene distrutto:**

1. **viene eseguito il corpo del distruttore della classe principale, se definito;**
2. **Quindi, vengono richiamati i distruttori delle classi secondarie, se definiti, nell'ordine inverso in cui queste compaiono nella dichiarazione dei membri della classe principale.**

# 18.2 Membro classe all'interno di classi (II)

**Se alcune classi secondarie possiedono costruttori con argomenti formali, anche per la classe principale deve essere definito un costruttore e questo deve prevedere una lista di inizializzazione**

```
// file record.h
#include "stringa.h"
class record                    // classe principale
{
    stringa nome, cognome;
public:
    record(const char n[], const char c[]);
    // …
};

// file record.cpp
record:: record(const char n[], const char c[])
  : nome(n), cognome(c)
{/*….*/}
//…

// file main.cpp
#include "record.h"
int main()
{
    record pers("Mario", "Rossi");
  //…
}
```

# 18.2 Membro classe all'interno di classi (III)

**Se alcune classi secondarie prevedono costruttori default, questi vengono richiamati.**
**Se tutte le classi secondarie hanno costruttori default, anche quello della classe principale può essere un costruttore default o mancare (viene usato il costruttore predefinito).**

```
// file stringa.h
class stringa
{  char* str;
public:
     stringa(const char s[]="");
//…
};

// file record.h
#include "stringa.h"
class record                    // classe principale
{
    stringa nome, cognome;
public:
    // …
};

// file main.cpp
#include "record.h"
int main()
{
    record pers;
  //…
}
```

# 18.3 Array di oggetti classe (I)

**Un array può avere come elementi oggetti classe: se nella classe sono definiti costruttore e distruttore, questi vengono richiamati per ogni elemento dell'array**

```
//file complesso.h
class complesso
{       double re, im;
public:
    complesso(double r = 0, double i = 0);
 //….
};

// file complesso.cpp
complesso:: complesso(double r, double i)
{re = r; im = i;}
/*…*/

// file main.cpp

#include<iostream>
#include "complesso.h"
using namespace std;
int main()
{
    complesso vc[3];
    for (int i=0; i<3; i++)
        vc[i].scrivi(); cout << endl;
    return 0;
}
```

```
(0, 0)
(0, 0)
(0, 0)
```

# 18.3 Array di oggetti classe (II)

**Inzializzazione esplicita**

```
// file main.cpp
#include<iostream>
#include "complesso.h"
using namespace std;
int main()
{   complesso vc[5] = {complesso(1.1,2.2),
                complesso(1.5,1.5), complesso(4.1,2.5)};
    for (int i=0; i<5; i++)
    {    vc[i].scrivi(); cout << endl;}

    return 0;
}
```

```
(1.1, 2.2)
(1.5, 1.5)
(4.1, 2.5)
(0, 0)
(0, 0)
```

**ATTENZIONE: se la lista di costruttori non è completa, nella classe deve essere definito un costruttore default, che viene richiamato per gli elementi non esplicitamente inizializzati.**

**ATTENZIONE: se l'array di oggetti è allocato in memoria dinamica, se nella classe è definito un costruttore, questo deve essere necessariamente default, in quanto non sono possibili inizializzazioni esplicite.**

# 18.4 Membri statici (I)

**Membri dati statici: modellano informazioni globali, appartenenti alla classe nel suo complesso e non alle singole istanze.**

```
// file entity.h
class entity{
   int dato;
public:
// ...
   entity(int n);
   static int conto;          //dichiarazione della var. conto
};

// file entity.cpp
#include "entity.h"
int entity::conto = 0;        //definizione della var. conto

entity::entity(int n)
{  conto++; dato = n;  }

// file main.cpp

#include<iostream>
#include "entity.h"
using namespace std;
int main(){
    entity e1(1), e2(2);
     cout << "Numero istanze: " << entity::conto << endl;
     return 0;
}
```

```
Numero istanze: 2
```

# 18.4 Membri statici (II)

**Funzione membro statica: operazione che non viene applicata ad una singola istanza della classe, ma svolge elaborazioni sulla classe in quanto tale.**

```
// file entity1.h
class entity1
{   int dato;
    static int conto;
public:
// ...
    entity1(int n);
    static int numero();     // Funzione membro statica
};
```

```
// file entity1.cpp
#include "entity1.h"
int entity1::conto = 0;

entity1::entity1(int n)
{  conto++; dato = n;  }

int entity1::numero()
{ return conto;}
```

```
// file main.cpp
//…
int main(){
     entity1 e1(1), e2(2);
    cout << "Num. istanze: " << entity1::numero() << endl;
    cout << "Num. istanze: " << e1.numero() << endl;
    return 0;
}
```

```
Num. istanze: 2
Num. istanze: 2
```

# 18.4 Membri statici (III)

**Una funzione membro statica non può accedere implicitamente a nessun membro non statico né usare il puntatore this.**

```
// file entity2.h
class entity2
{   int dato;
    int ident;
    static int conto;
public:
// ...
    entity2(int n);
    static int numero();     // Funzione membro statica
    static void avanzaConto(int n);
};
```

```
// file entity2.cpp
#include "entity2.h"
int entity2::conto = 0;

entity2::entity2(int n)
{  conto++; dato = n;  ident = conto;}

int entity2::numero()
{ return conto;}

void entity2::avanzaConto(int n)
{
    conto += n;
    ident = conto++;
  // ERRORE: invalid use of member `entity2::ident' in
  // static member function
}
```

# 18.5 Funzioni const (I)

**Funzione membro const: funzioni che non possono modificare l'oggetto a cui sono applicate.**
**Importanti perché ad oggetti classe costanti possono essere applicate solo funzioni membro costanti.**

```
// file complesso.h
class complesso
{
   double re, im;
public:
   complesso(double r = 0, double i = 0);
   double reale();
   double immag();
   //…
};

// file main.cpp
#include<iostream>
#include "complesso.h"
using namespace std;
int main()
{
 const complesso c1 (3.2, 4);
 cout << "Parte Reale " << c1.reale() << endl;
 //ERRORE: c1 e' costante

 return 0;
}
```

# 18.5 Funzioni const (II)

```
// file complesso.h
class complesso
{
 double re, im;
public:
 complesso(double r = 0, double i = 0);
 double reale() const;
 double immag() const;
//…
};

// file complesso.cpp
double complesso:: reale() const {return re;}
double complesso::immag() const {return im;}
//…

// file main.cpp

#include<iostream>
#include "complesso.h"
using namespace std;
int main()
{
   const complesso c1 (3.2, 4);
    cout << "Parte Reale " << c1.reale() << endl;
    return 0;
}
```

```
Parte Reale 3.2
```

# 18.5 Funzioni const (III)

**Il meccanismo di overloading distingue due versioni di una funzione che differiscono solo per la proprietà di essere costanti**

```
class complesso
{
  double re, im;
public:
  complesso(double r = 0, double i = 0):re(r),im(i){};
  double reale() const;
  double reale();
//…
};

// file complesso.cpp
double complesso:: reale() const
{cout << "[ chimata a complesso::reale()const ]" << endl;
return re;}

double complesso:: reale()
{cout << "[ chimata a complesso::reale() ]" << endl; return re;}

// file main.cpp
int main()
{   const complesso c1 (3.2, 4);
    complesso c2(7.8, 5);
    cout<< c1.reale() << endl;
    cout<< c2.reale() << endl;        //…      NOTARE L'USCITA
    return 0;
}
```

**[ chimata a complesso::reale()const ]**
**3.2**
**[ chimata a complesso::reale() ]**
**7.8**

# 18.7 Espressioni letterali

**Non è possibile definire espressioni letterali per i tipi definiti dall'utente. I costruttori possono comunque svolgerne la funzione**

```
// file complesso.h
class complesso
{
 double re, im;
public:
 complesso(double r = 0, double i = 0);
 complesso operator+(const complesso& x); //…
};

// file complesso.cpp
complesso complesso::operator+(const complesso& x)
{       complesso z(re+x.re,im+x.im);
        return z;
}
//….

// file main.cpp
//…
int main()
{   complesso c1 (3.2, 4), c2;
    c2 = complesso(0.3,3.0) + c1;
    c2.scrivi(); cout << endl;
//…
}
```

```
(3.5, 7)
```

# 18.8 Conversioni mediante costruttori

**Un costruttore che può essere chiamato con un solo argomento viene usato per convertire valori del tipo dell'argomento nel tipo della classe a cui appartiene il costruttore.**

```
// file complesso.h
class complesso
{
 double re, im;
public:
 complesso(double r = 0, double i = 0);
 complesso operator+(const complesso& x); //…
};


// file main.cpp
//…
int main()
{   complesso c1 (3.2, 4), c2;
    c2 = c1 + 2.5;
    // Il costruttore trasforma il reale 2.5 nel complesso (2.5,0)
    c2.scrivi(); cout << endl;
//…
}
```

```
(5.7, 4)
```

# 18.8 Operatori di conversione

**Operatore di conversione: converte un oggetto classe in un valore di un altro tipo.**

```
// file complesso.h
class complesso
{
 double re, im;
public:
 operator double();
 operator int();
 //…
};

// file complesso.cpp
complesso::operator double()
{ return re+im;}

complesso::operator int()
{ return static_cast<int>(re+im);}

// file main.cpp
//…
int main()
{   complesso c1 (3.2, 4);
    cout << (double)c1 << endl;
    cout << (int)c1 << endl;
//…
}
```

```
7.2
7
```

# 20. Classi per l'ingresso e l'uscita (I)

**Le librerie di ingresso/uscita forniscono classi per elaborare varie categorie di stream**

**Classe per l'ingresso dati**

```
class istream
{   // stato: configurazione di bit
public:
 istream(){…};
 istream& operator>>(int&);
 istream& operator>>(double&);
 istream& operator>>(char&);
 istream& operator>>(char* );
 int get();                                                    // (1)
 istream& get(char& c);                                        // (2)
 istream& get(char* buf, int dim , char  delim = '\n');        // (3)
 istream& read(char* s, int n)                                 // (4)
//…
};
```

**L'oggetto cin e un'istanza predefinita della classe istream.**

1. **La funzione *get()* preleva un carattere e lo restituisce convertito in intero**
2. **La funzione *get(c)* preleva un carattere e lo assegna alla variabile *c***
3. **La funzione *get(buf,dim,delim)* legge caratteri dallo stream di ingresso e li trasferisce in *buf* aggiungendo il carattere nullo finale finché nel buffer non vi sono *dim* caratteri o non si incontra il carattere *delim* (che non viene letto).**
4. **La funzione *read(s,n)* legge *n* byte e li memorizza a partire dall'indirizzo contenuto in s.**

# 20. Classi per l'ingresso e l'uscita (II)

**Classe per l'uscita dati**

```
class ostream
{   // stato: configurazione di bit
    // ….
public:
 ostream(){…};
 ostream& operator<<(int);
 ostream& operator<<(double);
 ostream& operator<<(char);
 ostream& operator<<(const char* );
 ostream& put(char c)
 ostream& write(const char* s, int n)
  ostream& operator<< (ostream& ( *pf )(ostream&));

//…
};
```

**Gli oggetti cout e cerr sono istanze predefinite della classe ostream.**

**La funzione *put()* trasferisce il carattere *c* sullo stream di uscita**
**La funzione *write()* trasferisce la sequenza di *n* caratteri contenuti in *s* sullo stream di uscita.**

# 20. Controllo del formato (I)

**Controllo del formato: può avvenire attraverso i manipolatori**

**Manipolatori: puntatori a funzione.**

**Gli operatori di lettura e scrittura sono ridefiniti per overloading in modo da accettare tali puntatori.**

**iostream**

**endl** \\ ostream& endl ( ostream& os );

inserisce '\n' e svuota il buffer

**cout << x << endl;**

**dec, hex, oct**

selezionano la base di numerazione

**cout << 16 << ' ' << oct << 16 << endl;**

**boolalpha**

inserisce o estrae oggetti di tipo booleano come nomi (true o false) piuttosto che come valori numerici

**cout << boolalpha << true << endl;**

**iomanip**

**setw(int)**

**numero minimo di caratteri usati per la stampa di alcune rappresentazioni (w sta per *width*)**

**setfill(char)**

**specifica quale carattere prependere per arrivare alla lunghezza desiderata (default: spazio bianco).**

**setprecision(int)**

**numero di cifre significative (parte intera + parte decimale) per la stampa di numeri reali**

**I manipolatori hanno effetto fino alla nuova occorrenza del manipolatore; eccetto setw() che ha effetto solo sull'istruzione di scrittura immediatamente successiva**

# 20. Controllo del formato (II)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(){
   double d = 1.564e-2;   int i;    bool b;
   cout << d << endl;
   cout << setprecision(2) << d << endl;
   cout << setw(10) << d << endl;
   cout << setfill('0') << setw(10) << d << endl;
   cin >> hex >> i; // supponendo di inserire 'b'
   cout << hex << i << '\t' << dec <<  i << endl;
   cin >> oct >> i; // supponendo di inserire “12”
   cout << oct << i << '\t' << dec <<  i << endl;
   cin >> boolalpha >> b; // supponendo di ins. “false”
   cout << b << boolalpha << ' ' << b << endl;
   cout << true << endl;
   return 0;
}
```

```
0.01564
0.016
     0.016
000000.016
b
b       11
12
12      10
false
0 false
true
```

# 20.2 Controlli sugli stream

**Stato di uno stream: configurazione di bit, detti *bit di stato*. Lo stato è corretto (*good*) se tutti i bit di stato valgono 0.**

- **Bit *fail* – posto ad 1 ogniqualvolta si verifica un errore recuperabile**
- **Bit *bad* – discrimina se l'errore è recuperabile (0) o meno (1)**
- **Bit *eof* – posto ad 1 quando viene letta la marca di fine stream**

**I bit possono essere esaminati con le funzioni seguenti, che restituiscono valori booelani:**

- **fail() – restituisce *true* se almeno uno dei due bit *fail* e *bad* sono ad 1**
- **bad() – restituisce *true* se il bit *bad* è ad 1**
- **eof() – restituisce *true se* il bit *eof* è ad 1**
- **good() – restituisce *true* se tutti i bit sono a 0**

**I bit possono essere manipolati via software con la funzione *clear()*, che ha per argomento (di default è 0) un valore intero che rappresenta il nuovo valore dello stato.**

**Possono essere usati i seguenti enumeratori:**

- ***ios::failbit* – corrisponde al bit *fail* ad 1**
- ***ios::badbit* – corrisponde al bit *bad* ad 1**
- ***ios::eofbit* – corrisponde al bit *eof* ad 1**

**Ogni valore dello stato può essere ottenuto applicando l'operatore OR bit a bit con operandi tali enumeratori.**

**NOTA BENE: Quando uno stream viene posto in una condizione, viene restituito il complemento del risultato della funzione *fail()*.**

# 20.3 Uso dei file

**Le classi *fstream*, *ifstream* e *ofstream* hanno una struttura simile a quella vista per gli stream di ingresso/uscita.**

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    ifstream input("nonEsiste.txt"); char in;
    cout << input.fail() << ' ' << input.bad() << ' '
        << input.eof() << ' ' << input.good() << endl;
    input.clear();
    cout << input.fail() << ' ' << input.bad() << ' '
        << input.eof() << ' ' << input.good() << endl;
    input.clear(ios::badbit);
    cout << input.fail() << ' ' << input.bad() << ' '
        << input.eof() << ' ' << input.good() << endl;
    input.clear();
    input.open("Esiste.txt");
    while (input>>in);
    cout << input.fail() << ' ' << input.bad() << ' '
        << input.eof() << ' ' << input.good() << endl;

    return 0;
}
```

```
1 0 0 0
0 0 0 1
1 1 0 0
1 0 1 0
```

# 20.4 Ridef. operatori ingresso/uscita(I)

**Il meccanismo di overloading permette di ridefinire gli operatori di lettura e di scrittura per i tipi definiti dall'utente.**

**ATTENZIONE: Tali operatori devono essere ridefiniti come funzioni globali.**

// file complesso.h

```
#include<iostream>
class complesso
{
    double re, im;
public:
    complesso(double r = 0, double i = 0){re=r;im=i;};
    double reale() const {return re;}
    double immag() const {return im;}
    //…
};

using namespace std;

ostream& operator<<(ostream& os, complesso z);
istream& operator>>(istream& is, complesso& z);
```

# 20.5 Ridef. operatori ingresso/uscita(II)

```
// file complesso.cpp
#include "complesso.h"

ostream& operator<<(ostream& os, const complesso& z)
{
    os << '(' << z.reale() << ','  << z.immag() << ')';
     return os;
}


istream& operator>>(istream& is, complesso& z)
{
    double re = 0, im = 0;
    char c = 0;
    is >> c;
    if (c != '(') is.clear(ios::failbit);
    else
    {
       is >> re >> c;
       if (c != ',') is.clear(ios::failbit);
       else
       {
           is >> im >> c;
           if (c != ')') is.clear(ios::failbit);
       }
    }
    z = complesso(re, im);
    return is;
}
```

# 20.5 Ridef. operatori ingresso/uscita(III)

```
#include<iostream>
#include "complesso.h"
using namespace std;

int main()
{
    complesso c1(1.0, 10.0);
    complesso c2(1.1, 10.1);
    cout << c1 << '\t' << c2 << endl;
    if (!(cin >> c2))
    {
        cerr << "Errore nella lettura di un numero
                    complesso" << endl;
        exit(1);
    }
    cout << c1 << '\t' << c2 << endl;

    return 0;
}
```

```
(1,10)  (1.1,10.1)
(2,12)
(1,10)  (2,12)
```

# 21 Preprocessore (I)

**Preprocessore: parte di compilatore che elabora il testo del programma prima dell'analisi lessicale e sintattica**

- **Può includere nel testo altri file**
- **Espande i simboli definiti dall'utente secondo le loro definizioni**
- **Include o esclude parti di codice dal testo che verrà compilato**

**Queste operazioni sono controllate dalle *direttive per il preprocessore* (il primo carattere è #)**

**Inclusione di file header. Due possibilità:**

- **Percorso a partire da cartelle standard**
  **#include <file.h>**
- **Percorso a partire dalla cartella in cui avviene la compilazione o percorso assoluto**
  **#include "file.h"**

**Esempio:**
**#include "subdir\file.h"**
**#include "c:\subdir\file.h"**

# 21 Preprocessore (II)

**Definizione di simboli con associato un valore**

```
#define KONST 42
int main()
{
    int x = KONST;          // come scrivere int x = 42;
   cout<<x<<endl;           //  stampa 42
}
```

**NB: Non bisogna abusare di questa tecnica. In questo caso specifico, avrebbe avuto più senso utilizzare la seguente soluzione:**

```
const int KONST = 42;

int main()
{
    int x = KONST;
   cout<<x<<endl;
}
```

**in quanto in questo secondo caso KONST ha anche un tipo (tipo intero), e pertanto il compilatore può fare tutti i controlli sui tipi nelle espressioni in cui compare KONST.**

**Pertanto la define per la definizione di costanti va utilizzata con parsimonia e solo in presenza di ulteriori considerazioni che ne giustifichino l'utilizzo.**

# 21 Preprocessore (III)

**Macro: Simbolo che viene sostituito con la sequenza di elementi lessicali corrispondenti alla sua definizione**

```
#define MAX(A,B) ((A)>(B) ? (A) : (B))
int main(){
    int y;
    y = 9 + MAX(2, 5);      // y = 9 + ( (2) > (5) ? (2) : (5) );
    cout<<y<<endl;          // stampa 14, come ci si aspetta
}
```

**ATTENZIONE però alle parentesi!!!!**
**Infatti, la macro definita come segue, senza parentesi:**

```
#define MAX(A,B) (A)>(B) ? (A) : (B)
int main(){
     int y;
     y = 9 + MAX(2, 5);     // y = 9 + (2) > (5) ? (2) : (5)
     cout<<y<<endl;         // y = 2, non 14!!!!
}
```

**farebbe sì che y alla fine valga 2 invece che 14:**

```
int main(){
     int y;
     y = 9 + (2) > (5) ? (2) : (5);

     // di conseguenza y alla fine vale 2 perchè l'operatore +
     // ha priorità maggiore dell'operatore ternario, che a sua
     // volta ha priorità minore dell'assegnamento:

     //  y = (9 + (2)) > (5) ? (2) : (5);
     //  y = ( 11 > 5 ? 2 : 5 );
     //  y = 2;
     cout<<y<<endl;         // stampa 2, non 14
}
```

# 21 Compilazione condizionale (I)

**Compilazione condizionale #if, #elif, #else, #endif**

**#if** *constant-expression*
*(code if-part)*
**#elif** *constant-expression*
*(code elif-parts)*
**#elif** *constant-expression*
*(code elif-parts)*
**#else**
*(code else-part)*
**#endif**

**I valori delle espressioni costanti vengono interpretati come valori logici ed in base a essi vengono compilati o no i frammenti testo (text) seguenti.**

**Esempio:**

```
#define DEBUG_LEVEL 1 // all'inizio del file

int main(){
#if DEBUG_LEVEL == 2
    cout<<i<<j;  // debug di i e j
#elif  DEBUG_LEVEL == 1
    cout<<i;      // debug della sola variabile i
#else
   // DEBUG_LEVEL == 0
   (nessuna cout)
#endif
  return 0;
}
```

# 21 Compilazione condizionale (II)

**FORME CONCISE**

**#ifdef identifier** per **#if defined identifier**
**#ifndef identifier** per **#if !defined identifier**

**ESEMPIO**

```
// file main.cpp

#define LINUX             // commentare questa riga e
// #define WINDOWS    //  scommentare questa
                          // per compilare sotto Windows

#include <cstdlib>
#include <iostream>
using namespace std;

int main()
  #ifdef LINUX     // equivale a '#if defined LINUX'
     system(" CLEAR");
  #elif defined WINDOWS
     system("CLS");
  #else
     cout<<"Sistema operativo non supportato"<<endl;
     exit(1);
  #endif
     return 0;
}
```

**DEFINE A RIGA DI COMANDO**

**Alternativamente, gli identificatori per il processore si possono definire invece che in main.cpp direttamente alla chiamata del compilatore:**

```
// Per compilare sotto LINUX
g++ -DLINUX main.cpp –o main.exe
// Per compilare sotto WINDOWS
g++ -DWINDOWS main.cpp –o main.exe
```

# 21 Compilazione condizionale (III)

**Altro utilizzo della compilazione condizionale:**

**evitare che uno stesso file venga incluso più volte in una unità di compilazione. Ogni file di intestazione, per esempio *complesso.h*, dovrebbe iniziare con le seguenti direttive**

---

```
// file complesso.h
#ifndef COMPLESSO_H
#define COMPLESSO_H
  // qui va il contenuto vero e proprio del file
  class complesso{
     double re, im;
  public:
    // funzioni della classe complesso
  };
#endif
```

---

```
// file main.cpp
#include "complesso.h"
#include <iostream>
#include "complesso.h"

int main(){
    complesso c;
    return 0;
}
```

l'aggiunta erronea di questo secondo include ora non causa più il problema di ridefinizione della classe *complesso*, perché la definizione della classe *complesso* viene inclusa solo la prima volta

---

# Appendice A – Classe di mem. dei vettori

**Caso 1: Vettore statico, avente lunghezza nota a TEMPO DI COMPILAZIONE.**

```
int vett1[10];   // vett1 deve essere def. fuori dai blocchi
```

**Caso 2: Vettore automatico, avente lunghezza nota a TEMPO DI COMPILAZIONE.**

```
const int DIM = 10;
int main(){
    int vett2[D];   // vettore automatico (quello «classico»)
}
```

**Caso 3: Vettore dinamico, avente lunghezza nota a TEMPO DI ESECUZIONE.**

```
int main(){
    int lun;
    cin>>lun;
    int *vett3 = new int[lun];    // vettore dinamico vero e proprio
}
```

**Caso 4: Vettore automatico, avente lunghezza nota a TEMPO DI ESECUZIONE.**

```
int main(){
    int lun;
    cin>>lun;
    int vett4[lun];
}
```

Come potremmo chiamare vett4? Vettore dinamico? No! In quanto esso ha classe di memorizzazione automatica (ossia viene allocato sullo stack). Qualcuno chiama vett4 vettore «**semi-dinamico**», per distinguerlo sia da quello automatico classico, che da quello dinamico vero e proprio.

# Appendice A – Classe di mem. dei vettori

**NB: Dentro una classe posso definire solo vettori dinamici veri e propri, oppure vettori automatici PURCHE' AVENTI LUNGHEZZA NOTA A TEMPO DI COMPILAZIONE.**

```
class Esempio1{
      int lun;
      int *vett;        // questo è consentito, purchè ovviamente
                        // il costruttore allochi memoria nello heap
public:
      ...
};


class Esempio2{
      int vett[10];        // questo è consentito
public:
      ...
};
```

**Quello che non si può fare è usare, dentro una classe, un vettore semi-dinamico, ossia con lunghezza nota a tempo di esecuzione!**

```
class Esempio3{
      int n;
      int vett[n];        // questo non è consentito
                          //  ed, infatti, non compila!
};
```

# Appendice B – Priorità degli operatori (1/2)

| Operatore (in grassetto) | Nome | Ass. |
|---|---|---|
| class-name :: member<br>namespace-name :: member | risolutore di visibilità | s |
| :: member | risolutore globale di visibilità | d |
| object . member | selezione | s |
| pointer -> member | dereferenziazione e selezione | s |
| array [ rvalue ] | indicizzazione | s |
| function ( actual-argument-list ) | chiamata di funzione | s |
| lvalue ++ | postincremento | d |
| lvalue -- | postdecremento | d |
| static_cast type <rvalue> | conversione di tipo | s |
| const_cast type <rvalue> | conversione const | s |
| sizeof object / sizeof (type) | dimensione di oggetto o di tipo | d |
| ++ lvalue | preincremento | d |
| -- lvalue | predecremento | d |
| ~ rvalue | complemento bit a bit | d |
| ! rvalue | negazione | d |
| - rvalue | meno unario | d |
| + rvalue | più unario | d |
| & lvalue | indirizzo | d |
| * rvalue | dereferenziazione | d |
| new type | allocazione | d |
| delete pointer | deallocazione | d |
| delete[] pointer | deallocazione di array | d |

# Appendice B – Priorità degli operatori (2/2)

| Operatore | Nome | Ass. |
|---|---|---|
| object .* pointer-to-member | selezione con punt. a membro | s |
| pointer ->* pointer-to-member | deref. e sel. con punt. a membro | s |
| rvalue * rvalue | moltiplicazione | s |
| rvalue / rvalue | quoziente | s |
| rvalue % rvalue | resto | s |
| rvalue + rvalue | somma | s |
| rvalue - rvalue | sottrazione | s |
| rvalue << rvalue | traslazione sinistra | s |
| rvalue >> rvalue | traslazione destra | s |
| rvalue < rvalue | minore | s |
| rvalue <= rvalue | minore o uguale | s |
| rvalue > rvalue | maggiore | s |
| rvalue >= rvalue | maggiore o uguale | s |
| rvalue == rvalue | uguale | s |
| rvalue != rvalue | diverso | s |
| rvalue & rvalue | AND bit a bit | s |
| rvalue ^ rvalue | OR esclusivo bit a bit | s |
| rvalue \| rvalue | OR bit a bit | s |
| rvalue && rvalue | AND logico | s |
| rvalue \|\| rvalue | OR logico | s |
| rvalue ? rvalue : rvalue | espressione condizionale | s |
| lvalue = rvalue | assegnamento | d |
| lvalue += rvalue | somma e assegnamento | d |
| lvalue -= rvalue | sottrazione e assegnamento | d |
| lvalue *= rvalue | moltiplicazione e assegnamento | d |
| lvalue /= rvalue | divisione e assegnamento | d |
| lvalue %= rvalue | resto e assegnamento | d |
| lvalue &= rvalue | AND bit a bit e assegnamento | d |
| lvalue \|= rvalue | OR bit a bit e assegnamento | d |
| lvalue ^= rvalue | OR esclusivo bit a bit e assegnamento | d |
| lvalue <<= rvalue | traslazione a sinistra e assegnamento | d |
| lvalue >>= rvalue | traslazione a destra e assegnamento | d |
| espression , expression | virgola | s |

(*fine delle slide*)