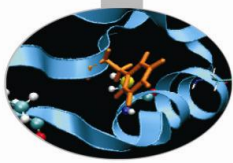
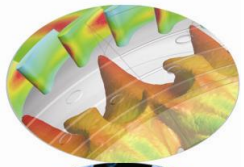
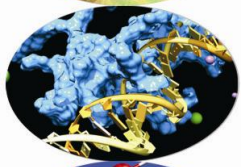
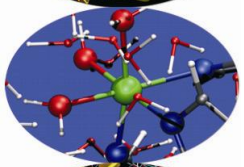
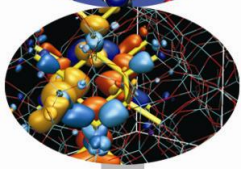
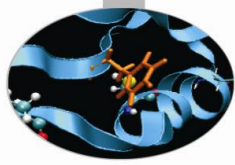


La programmazione ad oggetti

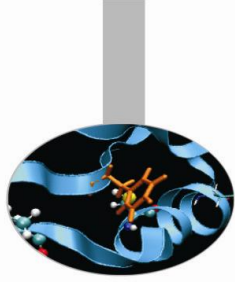





La programmazione ad oggetti e le classi:

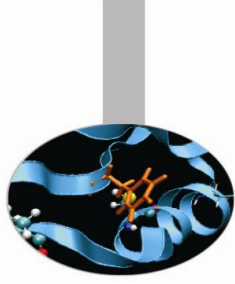


- **Introduzione ai concetti generali**
- **Paradigmi di programmazione**
- **Le classi: C-with-classes, C++**
- **Incapsulamento**
- **Ereditarietà**
- **Polimorfismo**
- **Commenti**

C-with-classes, C++



- L'aumentare delle dimensione dei codici (>100000 righe ad esempio i S.O) e dei dati da questi trattati, ha portato allo sviluppo dei metodi di programmazione. Si è passati dal singolo programmatore a gruppi di programmatori che interagiscono e lavorano su uno stesso progetto affrontandone separatamente aspetti distinti. Per supportare un tipo di programmazione che permettesse di dominare la complessità del codice, di strutturare il codice e di cooperare nella scrittura del codice in maniera naturale e pulita è stato pensato il C++.

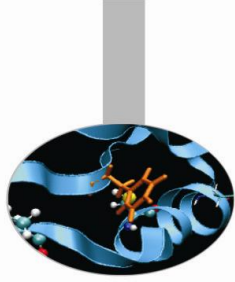


C-with-classes, C++

Il C++ è un linguaggio di programmazione *general purpose* che supporta il *data abstraction*, la programmazione *Object Oriented* e la programmazione generica.

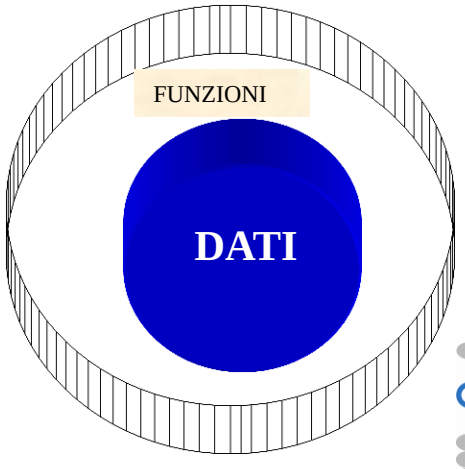
In generale si può affermare che la programmazione orientata agli oggetti ha un approccio bottom-up (individuazione di entità astratte e delle relazioni tra esse) mentre la programmazione procedurale segue un approccio top down (decomposizione funzionale).

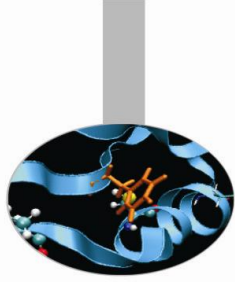
Ogni linguaggio che supporti una programmazione orientata agli oggetti, e quindi anche il C++, possiede strumenti generali per rendere chiara ed efficace questa potente tecnica di programmazione: **Incapsulamento dei dati, Polimorfismo, Ereditarietà.**



Incapsulamento

Ogni programma è costituito da due elementi fondamentali: il codice ed i dati. Il primo indica come svolgere le operazioni mentre i secondi sono gli oggetti cui applicare queste operazioni. L'incapsulamento dei dati è un meccanismo per cui, "conglobando" i dati e le operazioni su essi effettuate, permette di tutelare i dati da utilizzi scorretti o da alterazioni da parti di codice esterno non propriamente autorizzate. In questo senso il dato, o meglio il tipo di dato, va considerato come una coppia costituita dai possibili valori che un oggetto di quel tipo può assumere e dalle operazioni applicabili ad esso. La struttura dati è incapsulata nelle operazioni definite su di essa.





Incapsulamento (2)

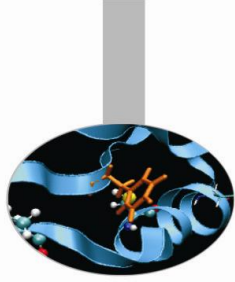
Il C++ supporta questo meccanismo offrendo degli strumenti per la definizione e la gestione di oggetti che incapsulano una struttura dati nelle operazioni definite su esse.

Con questi strumenti si crea una scatola nera, inaccessibile dall'esterno, che contiene i dati ed il codice atto a modificarli; questa scatola nera prende il nome di **CLASSE, l'istanza della classe è l' OGGETTO**. All'interno dell'oggetto il codice ed i dati possono essere pubblici o privati.

Per pubblici si intende che altre parti del programma possono accedere ad essi direttamente nonostante questi si trovino nell'oggetto.

Per privati si intende che l'accesso diretto è consentito solo ad altre parti dello stesso oggetto e anzi queste parti private dell'oggetto risultano completamente ignote fuori di esso.

Polimorfismo

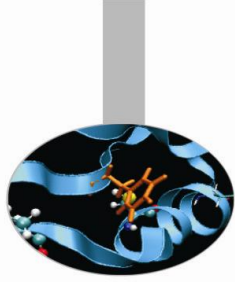


Definizione

È la funzionalità che permette di utilizzare una sola interfaccia per una classe di azioni.

È possibile creare una interfaccia generale ed unica che, con significati, implementazioni e contesti ben distinti, offre una certa funzionalità ad un gruppo di attività logicamente connesse. Il risultato primario è quello di una riduzione della complessità delegando al compilatore il compito di selezionare l'implementazione corretta necessaria nel contesto specifico in cui viene utilizzata quella determinata azione.

NOTA: il C++ è un linguaggio OO compilato e non interpretato, il polimorfismo è possibile sia in compilazione che in esecuzione.



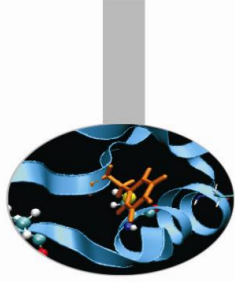
Ereditarietà

Definizione

Procedimento mediante il quale, seguendo un ordinamento gerarchico, un oggetto acquisisce le proprietà e le funzionalità di un altro oggetto.
Grazie all'ereditarietà ogni oggetto non deve ridefinire da zero le sue caratteristiche se sussiste una relazione con una classe di riferimento, ma partendo da quanto offerto dalla classe cui appartiene può arricchirsi di metodi suoi propri che ne definiscono la sua unicità.

L'ereditarietà multipla, peculiarità propria del C++, rende generalizzabile questo procedimento anche a più classi rispetto ad un medesimo oggetto.

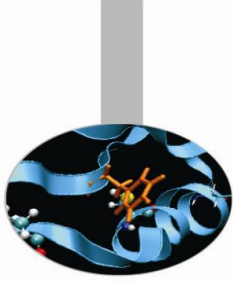
Esalta il riutilizzo del codice.



Commenti

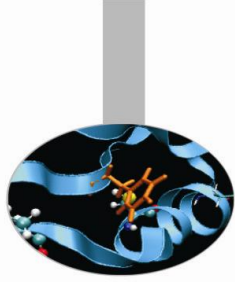
- Grazie alla programmazione ad oggetti un programma ora può essere scritto allo stesso modo in cui si assembla un aereo o si costruisce un edificio.

I rapporti tra le parti possono essere sfruttati in un processo interattivo e virtuoso.



Paradigmi di programmazione

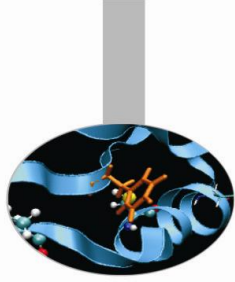
Introduzione



- Procedurale.
- Modulare o strutturato.
- Data abstraction.
- Object oriented.
- Generico.

Il C++ supporta questi paradigmi di programmazione nel senso che esistono degli strumenti che permettono di scrivere programmi secondo i concetti base che definiscono questi tipi.

Tuttavia nessuno di questi è a priori auspicabile o forzatamente preferibile rispetto agli altri in C++.



Programmazione Procedurale

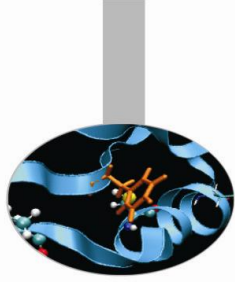
Decide which procedures you want; use the best algorithms you can find.

Il linguaggio supporta questo tipo di programmazione fornendo la possibilità di costruire procedure, routines e funzioni.

Un tipico esempio di questo è la creazione di una funzione che risolva un particolare task matematico.(esempio: radice(); inverti();…)

Strumenti forniti dal linguaggio:

- Procedure.
- Test e cicli.
- Librerie di funzioni.

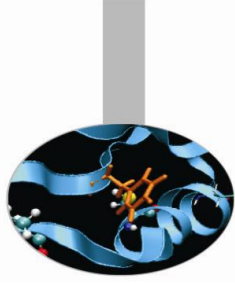


Esempio: stack procedurale

- È possibile implementare uno stack procedurale tramite un array e un indice all'ultimo dato.
- Possono essere implementate delle procedure apposite per manipolare la pila (push( ), pop( ) ed empty( )).
- Rimangono essenzialmente due grandi limiti:

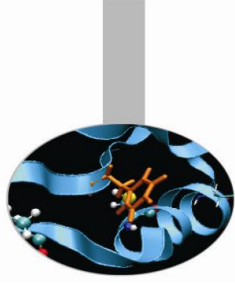
1. I dati contenuti nella pila sono accessibili e modificabili anche fuori dall'interfaccia delle funzioni scritte.
2. Per più stack e per tipi di dati diversi servono più repliche di codice ad hoc.

Di seguito tramite questo esempio, si vedrà come i diversi paradigmi di programmazione permettono di affrontare questi problemi.



Programmazione modulare o strutturata

- *Decide which modules you want; partition the program so that data is hidden within modules. (data-hiding principle)*
- Riflettendo lo sviluppo storico di nuove necessità nella scrittura di codici di maggiori dimensioni e che trattassero sempre più dati, l'enfasi è stata posta sul design del programma più che sulle singole procedure.
- Il C++ supporta questa programmazione permettendo di definire e compilare separatamente una interfaccia, una implementazione dell'interfaccia e un codice di utilizzo dell'interfaccia stesso. In linea di principio l'utilizzatore non è tenuto a conoscere i dettagli implementativi delle funzionalità che sono offerte dall'interfaccia.



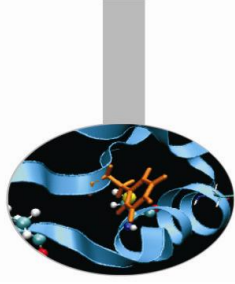
Programmazione modulare o strutturata(2)

Tornando all'esempio dello stack si potrebbe avere un file *stack.h (interfaccia o file dichiarativo)*, uno *stack.c (implementazione dell'interfaccia o file di definizione)* ed uno *user.c (file di utilizzo della struttura stack).*In questo modo i metodi di accesso ai dati sono forniti dalle funzioni descritte in stack.h.

I limiti che persistono sono:

- Non c'è ancora un modo naturale che limiti l'accesso diretto ai dati nè ad una loro modifica incontrollata.
- Per più stack e per tipi di dati diversi questa struttura va ridefinita ex novo

Programmazione modulare o strutturata(3)



Stack.h

Interfaccia o file di descrizione delle funzioni d'accesso (**push ( ), pop ( ), empty ( )**).

Stack.c

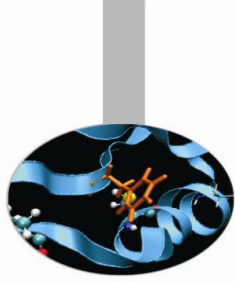
Implementazione dell' interfaccia.

Questo file contiene le definizioni (implementazioni) delle funzioni dichiarate nel file stack.h.

User.c

inclusione di stack.h

In questo file sarà contenuto il main e le chiamate alle funzioni di interfaccia per gestire una pila.

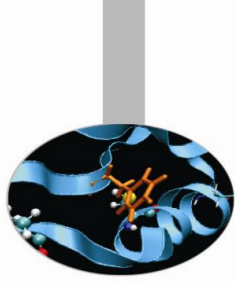


Data abstraction

Decide which types you want; provide a full set of operations for each type.

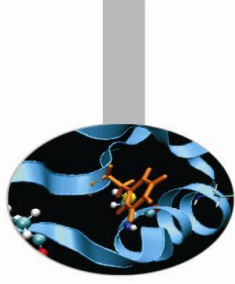
Questo tipo di programmazione, essendo di un ordine di complessità maggiore, risulta inutile per un unico oggetto di un solo tipo.

- Consiste nel definire nuovi TIPI (*abstract data* o *user-defined-type)* aventi un loro comportamento e stato dipendente in parte da come vengono definite le funzioni d'interfaccia e in parte da come abbiamo presentato il tipo rappresentativo.
- Quando servono dati di quel tipo si dice che il tipo viene istanziato.
- In C++ lo strumento utilizzato è la classe. Una classe è un nuovo tipo di dato, e come ogni tipo di dato, è costituita dalla coppia (dati, operazioni sui dati).



Data abstraction (2)

- Tornando all'esempio dello stack , con questo paradigma di programmazione verrà creato il nuovo tipo di dato stack.
- Utilizzando la definizione di un nuovo tipo di dato *stack,* inteso come dati ed operazioni per manipolare gli stessi, si attua automaticamente e in maniera "pulita" l'incapsulamento dei dati.
- Il grosso limite di questo strumento, peraltro basilare per una buona programmazione ed un buona strutturazione del programma, è la potenziale mancanza di flessibilità. La black box che viene definita non interagisce veramente con il resto del codice e non c'è modo di adattarla per nuovi usi se non modificandone la definizione stessa.

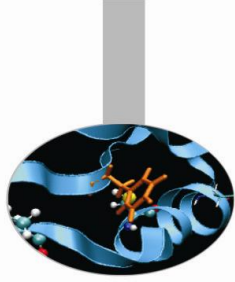


Programmazione ad Oggetti

Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance.

- La programmazione ad oggetti nasce dalla volontà di distinguere i tratti e le proprietà che caratterizzano una classe e di trarre vantaggio da questa caratterizzazione sfruttando l'ereditarietà.
- Gerarchia delle classi, classi astratte e polimorfismo si completano a vicenda al posto di essere mutuamente esclusive e rendono finalmente dinamica la classe.

Esempio: forme geometriche.



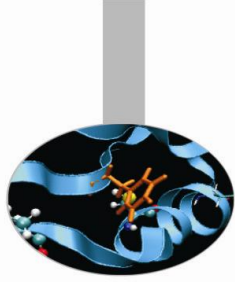
Programmazione Generica

Decide which algorithms you want; parametrize them so that they work for a variety of suitable types and data structures.

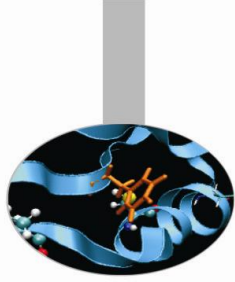
Per ampliare ulteriormente la generalità dei concetti espressi dal codice e permetterne quindi un più ampio riutilizzo modificandone il meno possibile, viene fornito come strumento la programmazione generica (*templates).*

Il concetto generale è che se un algoritmo (es. ordinamento) può essere espresso indipendentemente dai dettagli rappresentativi e questo può essere fatto con naturalezza, allora può essere scritto così.

Esempio: pila fatta non da caratteri o da interi ma da elementi qualsiasi. Algoritmi di ordinamento……



Classi I parte



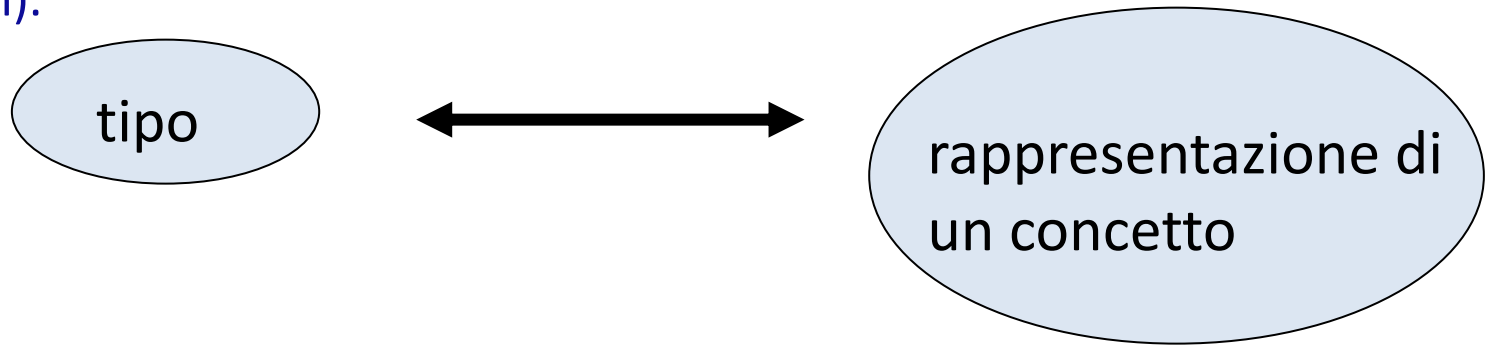
Class (1)

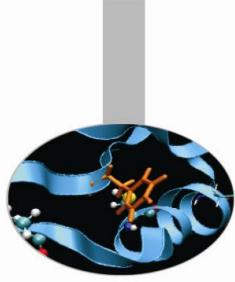
- Il costrutto:

```
class X {
    membri;
    funzioni membro (metodi);
};
```

definisce una classe.

- La definizione di una classe equivale alla definizione di un nuovo tipo: **user-defined type, ed il nome diventa pertanto una parola riservata con il proprio scope.**
- Questi nuovi tipi hanno le stesse potenzialità dei tipi predefiniti (o built-in).





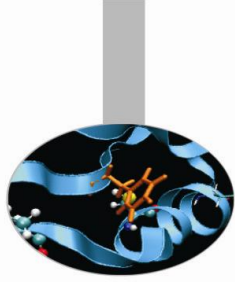
Class (2)

Implementare la definizione di un concetto che non ha una controparte diretta nei tipi built-in.

Ad esempio il tipo float inteso come insieme formato dai valori possibili e dalle operazioni ammesse su questi rappresenta una approssimazione del concetto di numero reale.

Associando un nuovo tipo ad un concetto si può ottenere un codice più comprensibile da un lato e più controllabile dall'altro. Usi scorretti della classe vengono rilevati dal compilatore a "compile time".

```
class myfirst{
            int a1;
            double a2;
            char c1;
            void f1();
            int f2(*int, *double, *char);
                        }; //opzionale lista oggetti
myfirst myobj1, obj2;
```

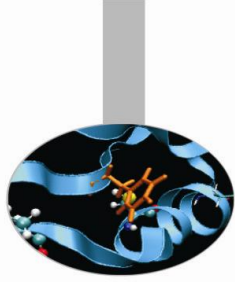


Class (3)

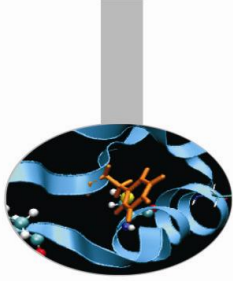
Le parole chiave `public` e `private` creano una distinzione netta tra chi può accedere ai membri e chi no all'interno del codice. l'invocazione di queste parole chiave è possibile in qualsiasi punto della definizione della classe e può essere ripetuto più volte.

```
class X {
      public:
            int     f1( );
            void    f2( );
      private:
            double        a,b,c;
      public:
            int     f3( );
      private:
            double        e;
};
```

Class(4)



- `public`: i membri, o le funzioni membro in questo caso, così dichiarati sono accessibili, visibili, a tutti. idealmente rappresentano l'interfaccia.
- `private`: i membri così dichiarati risultano accessibili, visibili, solo alle funzioni membro interne alla classe stessa.
- In questo modo ci si tutela dall'uso scorretto dei dati e di fatto il primo step del debugging, la localizzazione, viene fatta a compile time. L'interfaccia in questo modo è il solo mezzo per l'utente di manipolare, ma ora in modo del tutto canalizzato, i dati.

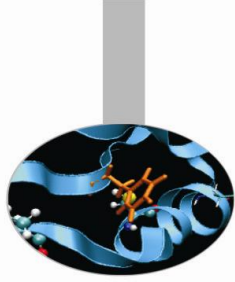


Esempio

Esempio sull'utilizzo di una classe per implementare una coda

```
#include <iostream>

class queue {
      int q[100];
      int sloc;
      int rloc;
    public:
      void init ( );
      void qput (int i);
      int    qget ( );
};
```

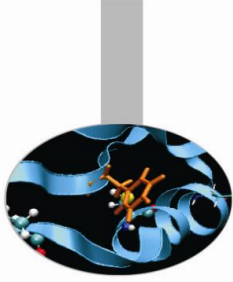
// dichiarazione della classe coda, qui vengono descritti solo i prototipi delle funzioni membro



Esempio

```
//implementazione delle funzioni membro pubbliche
void queue :: init ()
{
        rloc=sloc=0;
}
void queue :: qput (int i)  //utilizzo del risolutore di scopo (::)
        {
        if (sloc ==100) {
                cout << "full queue";
                return;
        }
        sloc++;
        q [sloc] = i;
}
int queue :: qget ( )
{
        if (rloc==sloc) {
                cout << " empty queue";
        return 0;
        }
        rloc++;
        return q[rloc];}
```

Esempio

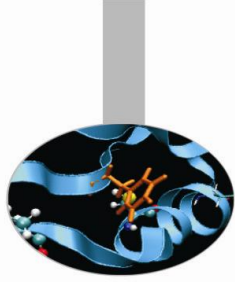


/* il main può essere scritto nello stesso file o separatamente. Qui si vuole fare notare l'uso della classe e dei suo membri. */

```
int main ( ) {
queue q1,q2;  //istanza della classe tramite la creazione
degli          oggetti q1 e q2.
q1.init ( );
q2.init( );
q1.qput (90);
q2.qput(69);
q1.qput (45);
q2.qput(30);
cout << "coda q1:" ;
cout << q1.qget () << "\t" <<q1.qget () <<"\n" ;
cout << "coda q2:" ;
cout << q2.qget () << "\t "<<q2.qget () <<"\n ";
return 0 ;
}
```

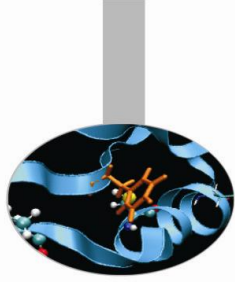
Ogni azione sulle code viene eseguita tramite l'interfaccia e mai direttamente sui dati. (incapsulamento)

Si noti il problema potenziale di init ( ). (costruttori).



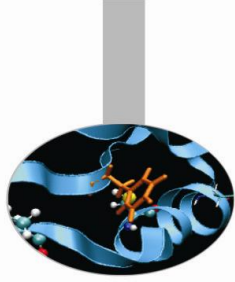
Costruttori e distruttori

- Il costruttore è una funzione void il cui compito è quello di inizializzare automaticamente l'oggetto al momento della sua creazione. Si distingue dalle altre funzioni essendo l'unica con lo stesso nome della classe. Possono essere implementati più modelli di costruttore per una stessa classe, il compilatore è in grado di identificare l'implementazione corretta per l'applicazione richiesta.
- Se non ne vengono forniti dal programmatore esplicitamente, vengono forniti, in qualche modo, implicitamente (default) dal compilatore.



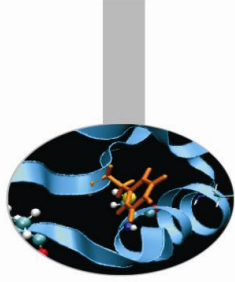
Costruttori e distruttori

- I costruttori di default non sempre funzionano correttamente, ad esempio nel caso di classi che contengono membri reference o const o nel caso di allocazione dinamica della memoria .
- Il distruttore è la funzione complementare ed è unico per ogni classe.
- Sintatticamente: `~nome_classe ( );`



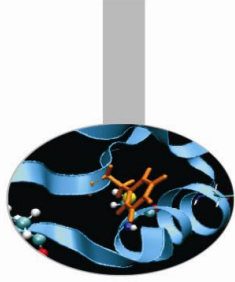
Esempio

```
//     counter.h
class contatore {
      int valore;
      public:
            contatore ( ); //costruttore senza
parametri
            contatore (int); //costruttore con
parametri
            ~contatore( );  //distruttore unico
            void init ( );
             void inc ( );
             void dec ( );
             int visualizza ( );
};
```



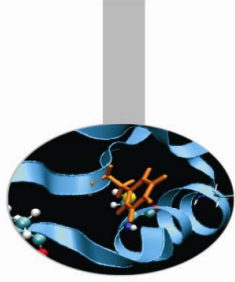
Esempio

```
//      counter.cpp
#include “counter.h”
 contatore :: contatore ( ) { valore =0; }
 contatore :: contatore ( int base) { valore = base; }
 contatore :: ~contatore ( ) { cout << “distruttore”; }
      void contatore :: init ( ) {       valore =0; }
       void contatore :: inc ( ) {       valore++; }
       void contatore :: dec ( ) {       valore--; }
       int contatore :: visualizza ( ) {return valore; }
```



Esempio

```
//     user_file.cpp
#include<iostream.h>
#include "counter.h"
int main ( )
{
      contatore c1; // costruttore invocato
      contatore c2 (69); // costruttore con
parametro invocato
      c1.inc ( );
      c2.inc( );
      cout << " c1 = " << c1.visualizza ( );
      cout << " c2 = " << c2.visualizza ( );
} /* a questo punto vengono invocati i distruttori, prima viene distrutto c2 e
poi c1.*/
```



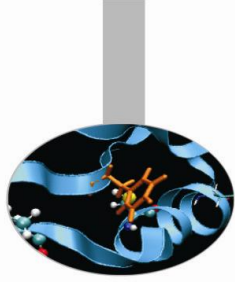
Funzioni Inline

- Per ovviare agli oneri in termini di Cpu legati alla chiamata di funzione, è possibile e utile, per funzioni molto semplici e utilizzate con frequenza elevata, definire la funzione come *inline*.
- In questo modo la funzione in fase di esecuzione non verrà chiamata, ma di fatto a compile time il suo codice verrà espanso nel punto dove essa è definita.
- A livello sintattico questo può essere fatto in due modi:

```
inline void contatore :: inc ( ) { valore++;}
```

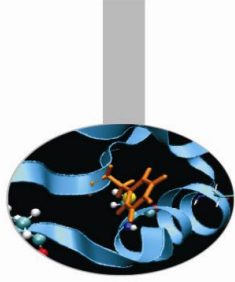
oppure direttamente senza la parola chiave:

```
void inc ( ) {valore++;}
```



Osservazione

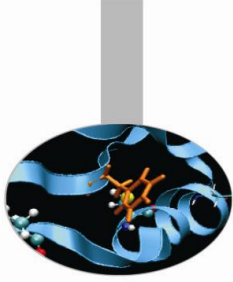
- Ogni qualvolta viene creata un'istanza di una classe per un oggetto, come già detto, viene creato uno spazio in memoria atto a contenere una copia di tutti i membri della classe.
- Questo non è vero per quei membri che siano dichiarati *static*.
- Un membro static non viene replicato per ogni oggetto, ma viene creata una sola copia comune a tutti gli oggetti della classe.
- La dichiarazione di un membro static comporta la sua successiva definizione affinché non venga segnalato errore a compile time.



Array di oggetti

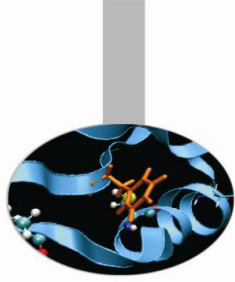
Un array di oggetti viene dichiarato nello stesso modo di un array di strutture e similmente si ottiene l'accesso ai suoi elementi tramite indicizzazione

```
#include <iostream.h>


class quadro
{
      double altezza;
      double larghezza;
public:
      void acquisisci_dimensioni();
      void caratteristiche();
};
```



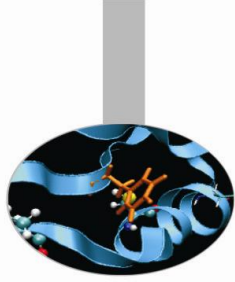
Array di oggetti

```
void quadro::acquisisci_dimensioni()
{
   cout << "digita le dimensioni del quadro in
cm.:<<endl;
   cout << "altezza: ";
   cin >> altezza;
   cout << "larghezza: ";
   cin >> larghezza;
}

void quadro::caratteristiche()
{
    cout << "\n altezza   = " << altezza;
    cout << "\n larghezza = " << larghezza;
    cout << endl<<endl;
}
```



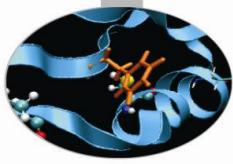
Array di oggetti

```
int main()
{
    int quantita;
    quadro galleria[100];

    cout << "inserisci il # dei quadri: "<<endl;
    cin >> quantita;

    for(int i = 0; i < quantita; i++)
        galleria[i].acquisisci_dimensioni();
        cout << endl<<"riassunto galleria"<<endl;
    for(int j = 0; j < quantita; j++)
    {
        cout << endl<< "opera #: " << j+1;
        galleria[j].caratteristiche();
    }
      return 0;
}
```



Esempio

```
inserisci il # dei quadri:
2
digita le dimensioni del quadro in cm.:
altezza: 12
larghezza: 12
digita le dimensioni del quadro in cm.:
altezza: 12
larghezza: 24
riassunto galleria:
opera #: 1
 altezza   = 12
 larghezza = 12

opera #: 2
 altezza   = 12
 larghezza = 24
```

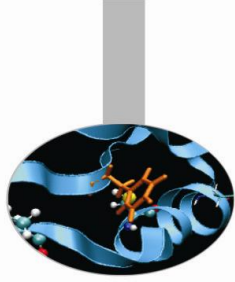


Puntatori ad oggetti

- E' possibile accedere direttamente ad un oggetto oppure ricorrendo ad un puntatore a quell'oggetto.
- Per accedere ai membri dell'oggetto, occorre utilizzare l'operatore freccia (→) in luogo dell'operatore punto.
- Per dichiarare un puntatore ad un oggetto la sintassi è analoga a quella utilizzata per qualunque altro tipo.

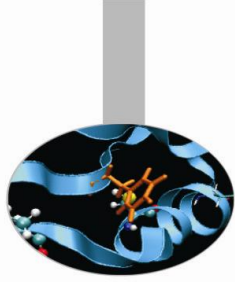
```
class punto {
      int ascissa;
      int ordinata;
   public:
      punto(int a,int b) {ascissa=a;ordinata=b;}
      void alto() { ordinata++;}
      void basso() { ordinata--;}
      void destra() {ascissa++;}
      void sinistra() { ascissa--;}
      int visual1() { return ascissa;}
      int visual2() { return ordinata;}
};
```

Esempio

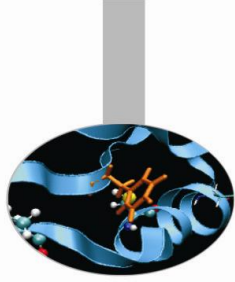


```
#include<iostream.h>
int main() {
char c;
punto p(0,0),*ptr;
ptr=&p;
   do { cout << "alto.........a"; cout <<
"basso.........b";
            cout << "destra........d"; cout <<
"sinistra......s";
            cout << "uscita........e"; cin >> c ;
     switch(c) {
       case 'a': ptr->alto(); break;
       case 'b': ptr->basso(); break;
       case 'd': ptr->destra(); break;
       case 's': ptr->sinistra(); break;
       case 'e': break;
              }
       cout<<ptr->visual1()<<","<<ptr->visual2()<<endl;
       } while(c!='e');
}
```

Passaggio di oggetti a funzioni



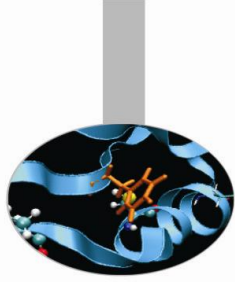
- Come ogni altra variabile di tipo built-in, anche un oggetto può essere passato ad una funzione.
- La modalità per valore è la procedura normale in C++. Questo passaggio per valore è anche detto per copia poiché viene passata una copia della variabile in questione. Continua a valere il concetto che qualunque modifica fatta sulla copia non ha alcun effetto sulla variabile passata.



Esempio

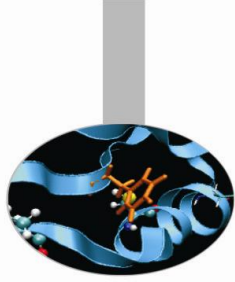
/* dichiarazione della classe e definizione dei metodi inline*/

```
#include   <iostream>
class myclass {
     int val;
public:
     myclass (int n);
     ~myclass ();
     void modifica_val(int n) {val=n;}
     int prendi_val() {return val;}
};
```



Esempio

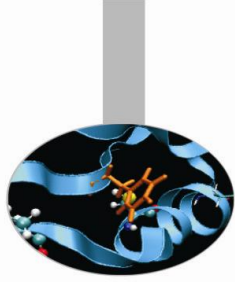
```
/* definizione del costruttore e del distruttore*/
myclass::myclass (int n)
{
      val = n;
      cout << "Costruito " << val << endl;
}
myclass::~myclass()
{
      cout << "Distrutto " << val << endl;
}
```



Esempio

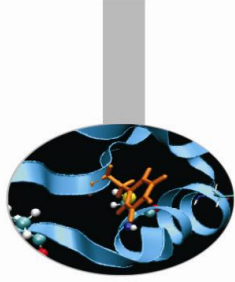
/* definizione della funzione cui passiamo un oggetto g(myclass)*/

```
void g(myclass obj);
void g(myclass obj)
{
obj.modifica_val(2);
cout << "valore locale: " << obj.prendi_val();
}
```



Esempio

```
int main()
{
 myclass dato(10); /*oggetto dato con val=10*/

 g(dato);        /* passaggio di oggetto a funzione*/
 cout << "valore nel main: " <<dato.prendi_val();
 return 0;
}
```



Esempio

OUTPUT:

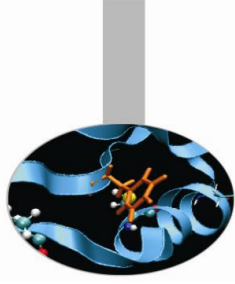
```
Costruito: 10

Valore locale: 2

Distrutto  2

Valore nel main: 10

Distrutto 10
```

Si noti come:

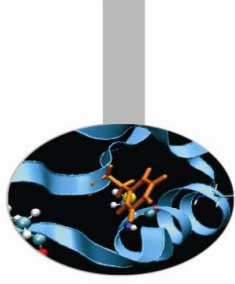
il costruttore non venga invocato all'ingresso nella funzione g() mentre il distruttore venga chiamato in uscita dalla funzione.

modifiche sull'oggetto effettuate all'interno della funzione non riguardano l'oggetto nel main, come per ogni tipo di variabile in C++.

Reference ad oggetti

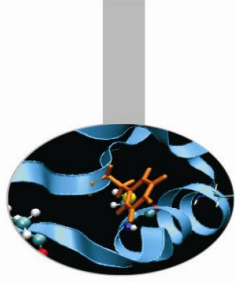


- Come per qualsiasi altro tipo di dato è possibile associare dei reference a degli oggetti.
- Non esistono particolari restrizione per l'utilizzo di reference a oggetti.



Assegnamento di oggetti

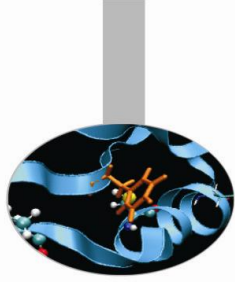
- Due oggetti del medesimo tipo, cioè appartenenti alla stessa classe possono essere assegnati tra loro.
- Per default viene fatta una copia bit a bit dei dati presenti nel termine a destra dell'uguale, cioè ne viene fatto un duplicato esatto.
- Assegnare un oggetto ad un altro rende identici i dati dei due oggetti, successive modifiche di uno di essi non alterano e non devono alterare i valori dei dati dell'altro. Le due entità rimangono separate.



Esempio

Esempio di assegnamenti ad oggetti, le definizioni sono quelle usate nell'esempio precedente.

```
int main()
{
 myclass obj1(10), obj2(20) ;
obj2 = obj1;          //assegnamento
obj1.modifica_val(90);
cout << "val2 dopo assign: " <<obj2.prendi_val();
cout << "val1 dopo mod: " <<obj1.prendi_val();
 return 0;
}
```



Esempio

OUTPUT:

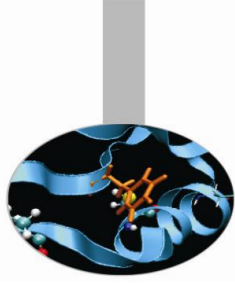
```
Costruito 10
Costruito 20
Val2 dopo assign: 10
Val1 dopo mod: 90
Distrutto 10
Distrutto 90
```

Si noti come:

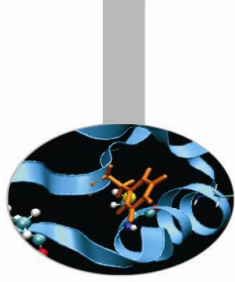
viene effettuata una copia bit a bit del right value nel left value.

i due oggetti sono indipendenti dopo l'assegnamento.

Restituzione di Oggetti

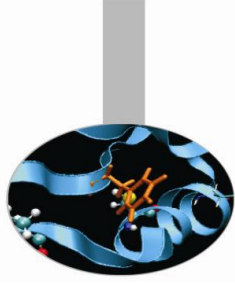


- Affinché una funzione restituisca un oggetto deve essere dichiarata come restituente il tipo classe dell'oggetto che si vuole restituire.
- La restituzione avviene tramite l'istruzione `return`.
- Quando un oggetto viene restituito, una variabile temporanea che contiene il valore da restituire viene creata. A restituzione avvenuta l'oggetto temporaneo viene distrutto.



Esempio

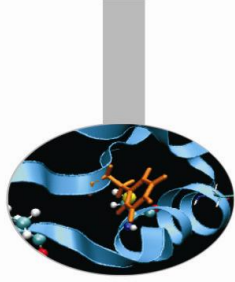
```
myclass f(int);
 int main()
{
     myclass  obj2(30);
     cout << "value of val in obj2 is: "<<endl;
     cout << "before calling f: ";
     cout << obj2.get_i() << endl;
     obj2= f(0);
     cout << "value of val in obj2 is: "<<endl;
     cout << "after calling f: ";
     cout << obj2.get_i() << endl;
     return 0;
}
```



Esempio

```
// implementazione della funzione
myclass f(int n)
{
      myclass obj(n);
      return obj;
}
```

Non presenta nessuna differenza rispetto ad altre funzioni che restituiscono un tipo.



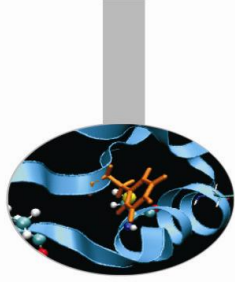
Esempio

OUTPUT

```
Costruito 30
value of val in obj2 is:
before calling f: 30
Costruito 0
Distrutto 0
Distrutto 0
value of val in obj2 is:
after calling f: 0
Distrutto 0
```

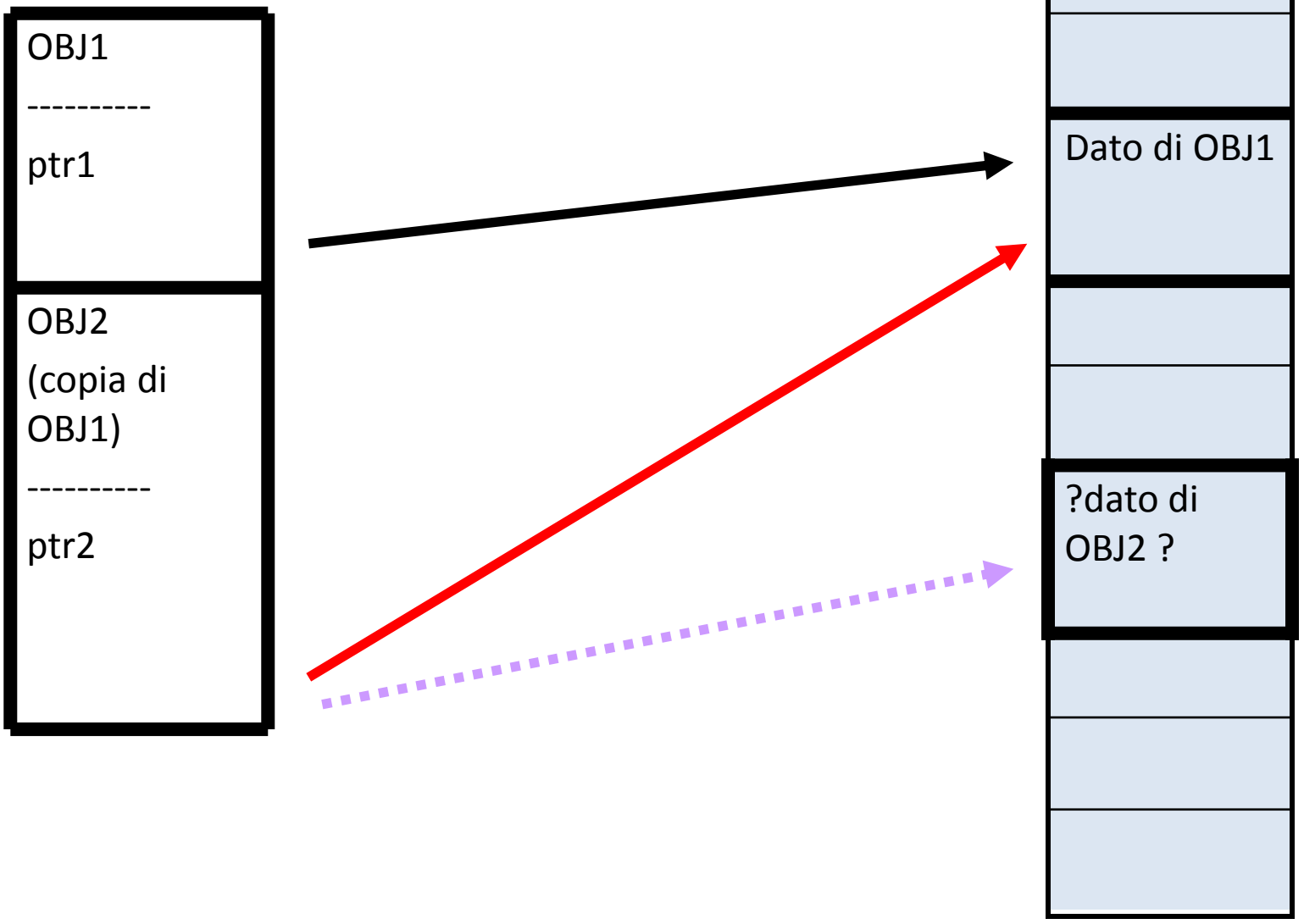
Si noti come:

Venga creata una copia per il ritorno e come per questa copia venga chiamato solo il distruttore.



Copie bit a bit

- L'uso di copie bit a bit di oggetti viene implicitamente fatto quando:
- Un oggetto è passato come parametro ad una funzione.
- Un oggetto è ritornato come variabile da una funzione.
- Un oggetto viene assegnato ad un altro.
- Un oggetto viene inizializzato tramite il valore di un altro oggetto.
- Questo tipo di procedura può causare sgraditi effetti collaterali quando gli oggetti trattati contengono membri di tipo puntatore e quando fanno uso di allocazione dinamica della memoria.


OBJ1
---
ptr1
OBJ2
(copia di OBJ1)
---
ptr2
Dato di OBJ1
?dato di OBJ2 ?