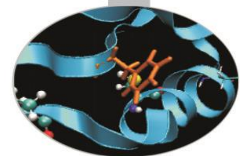
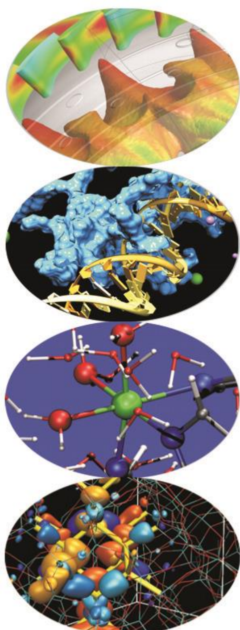
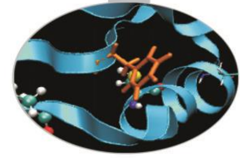
SuperComputing Applications and Innovation



Polimorfismo



CINECA SCAI SuperComputing Applications and Innovation

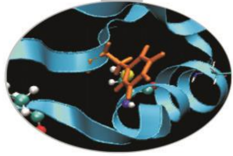


Indice

- Le funzioni virtuali e il puntatore alla classe base
- Le funzioni virtuali pure
- Classi astratte: interfaccia

CINECA

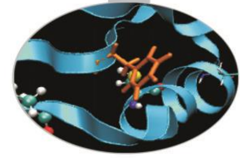
Il puntatore alla classe base e le funzioni virtuali



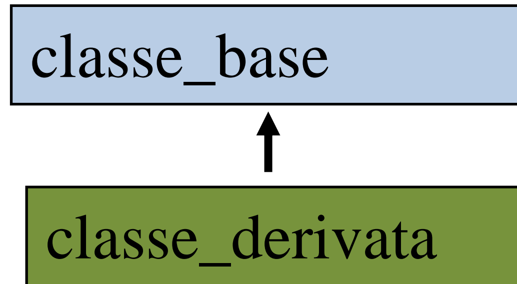
- Il puntatore alla classe base è uno degli elementi cardine del polimorfismo in esecuzione.

- I puntatori alla classe base e alle classi derivate da essa rappresenta un'eccezione alla regola generale per cui un puntatore ad un tipo non può essere utilizzato per manipolare oggetti di un altro tipo (STC).

Esempio

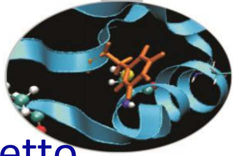


Puntatori a tipi derivati




```
/* in questo esempio si suppone di avere definito
una classe di base (classe_base) e di avere derivata
da essa un'altra classe (classe_derivata) */

classe_base  *pcb;  //puntatore ad un oggetto di tipo classe_base

classe_base B_obj;  // oggetto di tipo classe_base

classe_derivata D_obj;  // oggetto di tipo classe_derivata

pcb =  &B_obj; //ovvia

pcb = &D_obj;    /* istruzione sensata: ogni oggetto di classe derivata è
                    anche di classe base */
```

Osservazioni



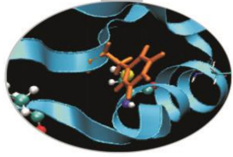
- La possibilità di utilizzare un puntatore alla classe base per manipolare un oggetto di tipo derivato è valida solo in questo senso e non nell'altro: un puntatore al tipo derivato non può manipolare oggetti definiti come appartenenti alla classe base.
- Il cast è un'operazione che per quanto corretta è da ritenersi deprecabile quantomeno per la leggibilità e la portabilità del codice.

Nell' esempio pcb può essere associato a D_obj ma volendo associare classe_derivata *pcd a B_obj dovremmo procedere così:

```
classe_derivata *pcd =
static_cast<classe_derivata*>(&B_obj);

/*potrebbe non avere senso (dipende dai dati ridefiniti
nella classe derivata)*/

classe_derivata *pcd =&B_obj; //ERRORE
```

Sarebbe bello se il linguaggio fosse in grado di stabilire autonomamente il tipo di un oggetto, manipolato tramite puntatore o reference, e utilizzare le implementazioni coerenti

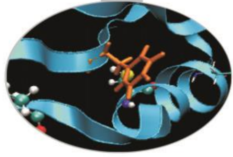
SuperComputing Applications and Innovation



Esempio(2)

```
// pointers to base class
#include <iostream.h>
class CPolygon {
      protected:
            int width, height;
      public:
            void set_values (int a, int b)
                  { width=a; height=b; } };

class CRectangle: public CPolygon {
      public: int area (void)
                  {return (width * height);} };

class CTriangle: public CPolygon {
      public: int area (void)
                  { return (width*height / 2); } };
```

SuperComputing Applications and Innovation

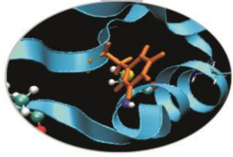


Esempio(2)

```
int main () {
      CRectangle rect;  //oggetti classe derivata
      CTriangle trgl;
      CPolygon * ppoly1 = &rect;  //nota (classe base)*
      CPolygon * ppoly2 = &trgl;
      ppoly1->set_values (4,5);
      ppoly2->set_values (4,5);
      cout << rect.area() << endl;
      cout << trgl.area() << endl;
      return 0; }
```

Questo utilizzo dei puntatori è solo una parte del meccanismo fornito dal C++ per mettere in atto il polimorfismo in esecuzione. L'altro elemento è fornito dalle funzioni virtuali.

Le funzioni virtuali



- Una funzione virtuale è una funzione membro di una classe dichiarata in maniera particolare la cui definizione (implementazione) può mutare all'occorrenza nelle varie classi derivate.

- Dichiarazione di una funzione membro virtuale:

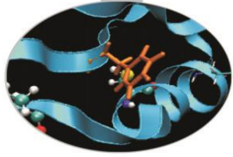
virtual void mostra_dato ( );

- Ricorrendo ad un puntatore alla classe base, si attiva il meccanismo di polimorfismo in fase di esecuzione. A questo punto alla chiamata della funzione virtuale tramite il puntatore all'oggetto il compilatore C++ è in grado di identificare l'implementazione corretta per quella chiamata.

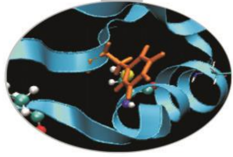
- Di default nella classe derivata si utilizza la ridefinizione, se esiste, della funzione dichiarata virtual nella classe base. Altrimenti si usa quella fornita dalla classe base.

- Una classe contenente delle funzioni virtuali è detta classe polimorfa.

Esempio



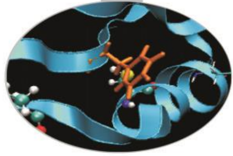
```
#include <iostream>
using namespace std;
class base  {
      public:
            virtual void mostra ( )
                  { cout <<   “classe base” <<
endl;}
            };
class derivata_1 :  public base {
      public:
            void mostra ( )
                  { cout << “prima derivazione”;}
            };
```



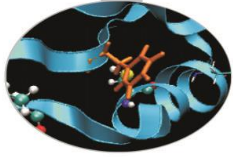
Esempio

```
int main ( ) {
        base base_obj;
        base *pb;
        derivata_1 derivata_obj;

        pb = & base_obj;
        pb → mostra ( );          /* accede alla funzione
                                  definita nella classe base*/
        pb = & derivata_obj;
        pb → mostra ( );                /* utilizza la funzione
    definita                            nella classe derivata*/
return 0;}
```

Esempio



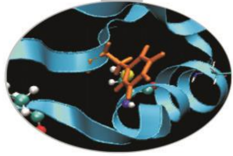
```
include <iostream>
class punto { //classe base, 1D
private:
double x; // unica coordinata, 1D
public:
punto(double a = 0) { x = a; } // costruttore
virtual void posizione() const { cout << x; } /* stampa la
                                               coordinata*/
};
class punto2d: public punto { // classe derivata 2D
private: // eredita la coordinata x da punto 1D
double y; // altra coordinata
public:
punto2d(double a = 0, double b = 0): punto(a){y = b;}
virtual void posizione() const {
punto::posizione(); // usa la funzione della classe base
cout << " " << y; // altra coordinata
}
};
```



Esempio

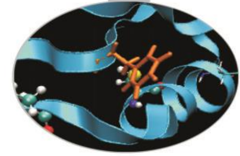
```
class punto3d: public punto2d { //3D
private: // eredita x, y da punto2d
double z; // z
public:
punto3d(double a = 0, double b = 0, double c = 0)
: punto2d(a, b){ z = c; }
virtual void posizione() const {
punto2d::posizione(); // usa quella di punto2d
cout << " " << z; // stampa anche la z
}
};
```

Esempio

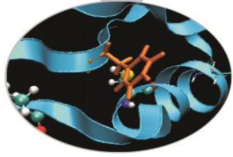


```
void func(const vector<punto*>& v) {
for (int i = 0; i < v.size(); i++) {
v[i]->posizione();
cout << '\n';
}
}

//MAIN
int main() { // test what is printed out
punto a(3.2);
punto2d b(2.8, 6.9);
punto3d c(8.7, 2.7, 1.0);
vector<punto*> v(3);
v[0] = &a;
v[1] = &b;
v[2] = &c;
func(v);
}
```

Osservazioni

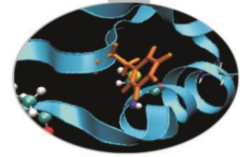


- La determinazione dell'implementazione da utilizzare avviene run time basandosi sul tipo puntato dal puntatore utilizzato per manipolare la funzione membro virtuale.
- La chiamata alla funzione virtuale può avvenire anche tramite il metodo convenzionale dell'operatore punto (**.**). Tuttavia questa chiamata ignora le proprietà polimorfiche della funzione.
- OVERLOAD $\neq$ FUNZIONI VIRTUALI. Le funzioni soggette ad overloading devono differire per numero e/o tipo di argomenti, mentre le funzioni virtuali ridefinite devono necessariamente essere identiche tra loro. Inoltre una funzione virtuale deve essere un membro e non un friend.
- Le funzioni virtuali possono essere dei distruttori ma non dei costruttori.



Esempio

```
class Base {
    public:
    virtual void show(){cout<<endl<<oggetto della classe Base''};
}
class Derivata1:Base {
    public:
    void show(){cout<<endl<<``oggetto della classe Derivata1''};
}
class Derivata2:Base {
    public:
    void show(){cout<<endl<<``oggetto della classe Derivata2''};
}
```



Esempio

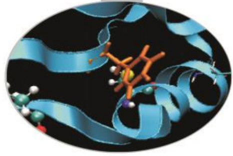
```
int main(){
     Base *pb,b;
     Derivata1 d1;
     Derivata2 d2;
     int choice;
     cout<<''Quale oggetto vuoi visualizzare?''
     cout<<''0 Base''<<endl;
     cout<<''1 Derivata1''<<endl;
     cout<<''2 Derivata2''<<endl;
     cin>>choice;
     switch(choice) {
     case(0):
           pb=&b;
           break;
     case (1):
           pb=&d1;
           break;
     case (2):
           pb=&d2;
           break;}
     pb->show();
     }
```

Late binding

Quale oggetto vuoi visualizzare?
0 Base
1 Derivata1
2 Derivata2
>> 2

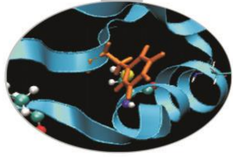
oggetto della classe Derivata 2

Esempio

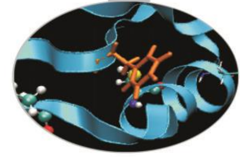


```
int main(){
    Base *pb,b;
    Derivata1 d1;
    Derivata2 d2;
    int choice;
    cout<<''Quale oggetto vuoi visualizzare?''
    cout<<''0 Base''<<endl;
    cout<<''1 Derivata1''<<endl;
    cout<<''2 Derivata2''<<endl;
    cin>>choice;
    switch(choice) {
    case(0):
          b.show();
          break;
    case (1):
          d1.show();
          break;
    case (2):
          d2.show();
          break;}
    }
```

Esempio



```
class Base {
    public:
    void show(){cout<<endl<<‘’oggetto della classe Base’’};
}
class Derivata1:Base {
    public:
    void show(){cout<<endl<<‘’oggetto della classe Derivata1’’};
}
class Derivata2:Base {
    public:
    void show(){cout<<endl<<‘’oggetto della classe Derivata2’’};
}
```



Esempio

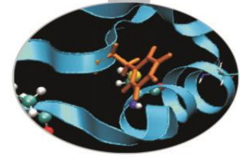
```
int main(){
    Base *pb,b;
    Derivata1 d1;
    Derivata2 d2;
    int choice;
    cout<<''Quale oggetto vuoi visualizzare?''
    cout<<''0 Base''<<endl;
    cout<<''1 Derivata1''<<endl;
    cout<<''2 Derivata2''<<endl;
    cin>>choice;
    switch(choice) {
    case(0):
        pb=&b;
        break;
    case (1):
        pb=&d1;
        break;
    case (2):
        pb=&d2;
        break;}
    pb->show();
    }
```

Upcasting e Static binding

Quale oggetto vuoi visualizzare?
0 Base
1 Derivata1
2 Derivata2
>> 2

oggetto della classe Base

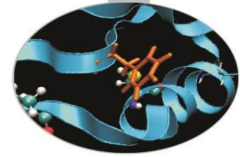
Gerarchia delle funzioni virtuali



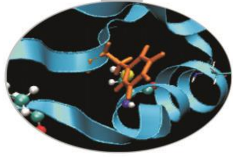
- Le funzioni virtuali dichiarate nella classe base vengono ereditate dalle classi derivate come tali per qualsiasi livello gerarchico.

- La funzione virtuale può essere ridefinita in maniera conveniente in ogni sottoclasse derivata come mostrato nell'esempio. Ogni sottoclasse, in assenza di una ridefinizione, fa riferimento alla classe più prossima nella scala gerarchica.

Costruttori e distruttori virtuali

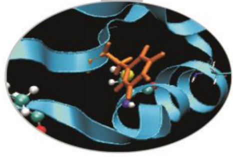


- Come discusso nel modulo precedente il costruttore della classe derivata deve contenere nella lista di inizializzazione il costruttore della classe base qualora presente
- Il distruttore virtuale è un distruttore comune che però ha anche la caratteristica di essere dichiarato **virtual**
- **Ha ovviamente senso solo qualora si stia manipolando memoria dinamica.**



Esempio

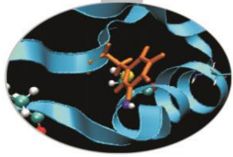
```
#include <iostream>
class base  {
      double *pd;
      public:
      base( ) {
       pd=new double[10];
       cout <<" new 10 double in base" << endl;}
      ~base(){
       delete [] pd;
       cout <<"distrutti 10 double in base\n";}

            };
```



Esempio

```
class derivata :  public base {
        char *pc;
  public:
   derivata(): base() {
    pc= new char [22];
            { cout << "new 22 char in derivata";}
  ~derivata(){
    delete [] pc;
    cout <<"distrutti 22 char in derivata\n";}
      };
```

Esempio



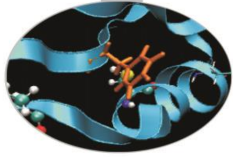
```
/*il codice*/
base *pb= new derivata;
delete pb;
/*produce questo:
   new 10 double in base
   new 22 char in derivata
   distrutti 10 double in base
*/
```

E’ necessario dichiarare il distruttore come virtuale nella classe base:

```
virtual ~base(){
        delete [] pd;
        cout <<“distrutti 10 double in base\n”;}
```

lasciando tutto il resto inalterato si ottiene:

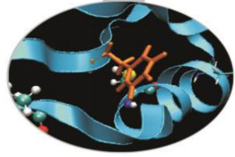
```
      new 10 double in base
      new 22 char in derivata
      distrutti 22 char in derivata
      distrutti 10 double in base
```



Classi astratte: interfaccia

- La classe base stabilisce l'interfaccia generale che caratterizza ogni oggetto, ma lascia libertà sul metodo.
- In realtà le potenzialità delle funzioni virtuali e le motivazioni dell'importanza del concetto di interfaccia si spiegano meglio in presenza di programmi voluminosi. Tuttavia qui vengono forniti gli elementi atti a comprenderne i meccanismi.
- Possono esistere concetti "astratti" quali ad esempio quello di forma che precludono la possibilità di definire oggetti di una classe associabile con quel concetto. Pertanto viene fornita in C++ la possibilità di definire classi che rappresentino concetti astratti, cioè classi per cui non sia possibile generare un'istanza ma da cui sia naturale derivare sottoclassi istanziabili cioè dotate di oggetti di quella classe.

Classi astratte

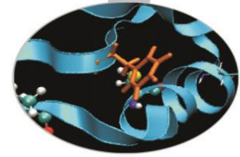


- Una funzione virtuale pura è una funzione che non ha definizione nella classe in cui è dichiarata; essa può essere solamente definita nelle classi derivate.
- Una classe che contiene almeno una funzione virtuale pura viene detta classe astratta.
- Una funzione virtuale pura viene dichiarata così:

```
virtual tipo Nome_funzione (lista_parametri)= 0 ;
```

- Dal momento che molte classi richiedono un corretto metodo di pulizia della memoria, e visto che una classe astratta a priori deve fornire un'interfaccia per questo scopo, solitamente è buona regola definire un distruttore virtuale nella classe base. La sua implementazione eventuale nella classe derivata dovrà essere adeguata al contesto: `virtual ~nome_classe_base () {};`

Classi astratte



Forma *

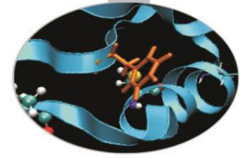
Cerchio

Triangolo

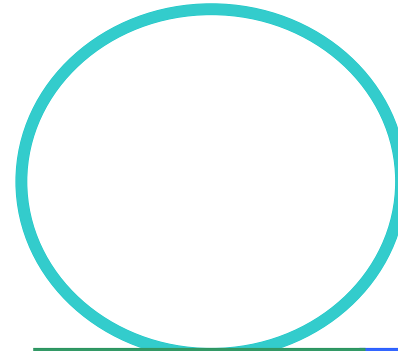
Composta

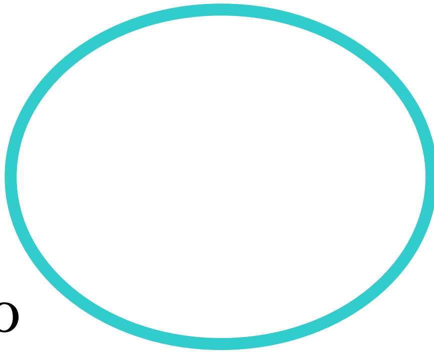
Rettangolo

Classi astratte

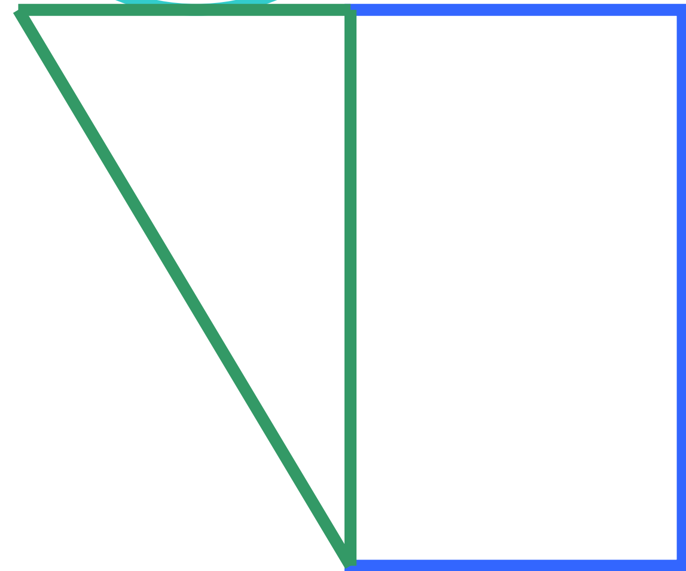


triangolo





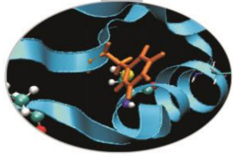
cerchio



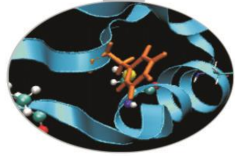
rettangolo

Figura_composta

Classi astratte



```
#include <iostream.h>

/* la classe base astratta provvede all'interfaccia
comune*/

class forma {

protected:

      double area;

public:

       virtual double calcola_area ( ) = 0;
       virtual void get_area()=0 ;

       virtual char get_type()=0;

       virtual void show()=0;};
```

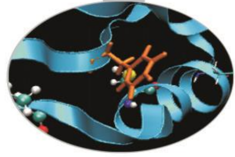


Classi astratte

```
/****************** TRIANGOLO**********************/
class triangolo: public forma {
   private:
        double base, altezza;
        char tipo ;
   public:
      triangolo(double i, double j)
       {base=i; altezza=j ;tipo ='t';} /*costruttore*/
      void show( ) {
      cout << endl<< " elemento di tipo: " << get_type()
                                          <<endl;
      cout <<  " dimensione base : " << base <<endl;
      cout <<  " dimensione altezza :" << altezza <<endl;
      cout <<  " area triangolo: " << area << endl;  }
      double calcola_area (){ area = (base * altezza)/2;
                             return area;}
      void get_area (){ cout << "l'area vale : " <<
                                        area << endl;}
      char get_type(){return tipo;}};
```

SuperComputing Applications and Innovation

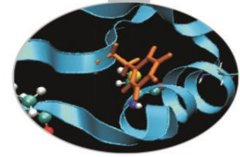
Classi astratte



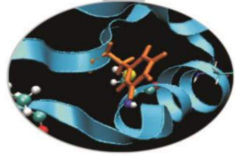
```
/*********** RETTANGOLO*********************/
class rettangolo : public forma {
 private:     double base, altezza;
              char tipo;
 public:
rettangolo(double i, double j){base=i; altezza=j ;tipo='r';}
void show( ) {
       cout << endl<< " elemento di tipo: " << get_type () <<
endl;
       cout <<  " dimensione base : "    << base <<endl;

       cout <<  " dimensione altezza :" << altezza <<endl;

       cout <<  " area rettangolo: " << area << endl;}
       double calcola_area (){ area = (base * altezza) ;return
                                                     area;}
       void get_area (){ cout << "l'area vale : "<< area <<
endl;}
char get_type(){return tipo;}};
```

CINECA

Classi astratte

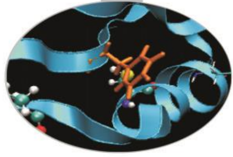


```
/***************************CERCHIO********************/
class cerchio : public forma {
   private:
       double raggio;
       char tipo;
   public:
       cerchio(double i){raggio = i ;tipo ='c';}
/*costruttore*/
       void show( ) {
       cout << endl << " elemento di tipo: " << get_type () <<
endl;  cout <<  " dimensione raggio: " << raggio <<endl;
       cout <<  " area cerchio: " << area << endl;    }
       double calcola_area (){ area = raggio * raggio * 3.1415;
return area;}
       void get_area (){ cout << "l'area vale : " << area <<
endl;}
       char get_type(){return tipo;}
};
```



Classi astratte

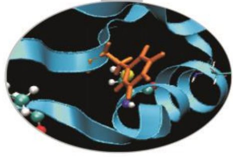
```
int main ( ){
        double area_figura=0;
        forma *pf = 0;
        forma  *figura_mista[3];
        triangolo T(10,20);
        rettangolo R(10,20);
        cerchio C(10);
        pf = &T;
        figura_mista[0] = pf;
        pf = &R;
        figura_mista[1] = pf;
        pf = &C;
        figura_mista[2] = pf;
```



Classi astratte

```
cout << "elementi presenti nella figura composta: "<<endl;
for(int i=0; i < 3; i++)  {

      area_figura += figura_mista[i] ->calcola_area();
      figura_mista[i] -> show();
            }
cout << endl << "area figura composta: " <<
area_figura < << endl << endl;
return 0;
}
```

Esempio



```
elementi presenti nella figura composta:
 elemento di tipo: t
 dimensione base : 10
 dimensione altezza :20
 area triangolo: 100
 elemento di tipo: r
 dimensione base : 10
 dimensione altezza :20
 area rettangolo: 200
  elemento di tipo: c
 dimensione raggio: 10
 area cerchio: 314.15
 area figura composta: 614.15
```