La programmazione orientata agli oggetti ed il linguaggio C++

Vitoantonio Bevilacqua
bevilacqua@poliba.it

Sommario. Il presente capitolo è ancora una bozza non controllata e si riferisce alle lezioni del corso di Fondamenti di Informatica e Laboratorio di Informatica relative alla programmazione orientata agli oggetti.

Parole chiave: incapsulamento, ereditarietà, polimorfismo a compile time ed a run time, costruttori e distruttori, overloading degli operatori ecc. ecc.

Ringraziamenti. Il presente capitolo è stato scritto anche grazie al prezioso contributo degli studenti Luigi Pansini, Antonio Pappalardo, Flavio Palmieri , Pasquale Bonasia.

1 Introduzione

OOP, Programmazione Orientata agli Oggetti

Per *Oggetto* si intende una entità che contiene (incapsula) un insieme di attributi (o dati o *variabili*) e comprende un insieme di operazioni (o *metodi* o funzioni) che manipolano i dati. Le Proprietà dei linguaggi di programmazione ad oggetti sono:

1.INCAPSULAMENTO
2.EREDITARIETA'
3.POLIMORFISMO o OVERLOADING

1. INCAPSULAMENTO

E' la proprietà costitutiva degli oggetti. Incapsulare vuol dire nascondere il funzionamento interno di una parte di programma. Quello che nascondiamo all'esterno sono variabili e metodi di un oggetto (information hiding).

esempiodiincapsulamento:datiefunzioniinunoggetto

```
// un oggetto è una istanza di una classe
// include la necessaria intestazione <iostream>
#include <iostream>
// l'istruzione using informa il compilatore che intende utilizzare il namespace standard
using namespace std;
// i namespace creano delle regioni di dichiarazione
// nel namespace standard viene dichiarata l'intera libreria standard del C++
#define SIZE 100
// Creazione della classe pila
// una classe può contenere parti private e parti pubbliche
// in generale tutti gli oggetti dichiarati all'interno di una classe sono privati
class pila {
private: //private è pleonastico (si può omettere)
        int PILA[SIZE];
        int puntatore_pila;
        // le variabili PILA e puntatore_pila si chiamano variabili membro e sono private
        // questo significa che non sono visibili da nessun'altra funzione che non sia
```

```
        // un membro della classe
public:
        // tutte le variabili e le funzioni definite dopo public sono pubbliche ovvero
        // possono essere utilizzate da tutte le altre funzioni del programma
        // limitare le variabili pubbliche
        void inizializza();
        void push(int);
        int pop();
        // inizializza, push e pop sono chiamate funzioni membro poichè membri della classe pila
        // solo le funzioni membro possono accedere alle variabili private
        // inizializza, push e pop sono chiamate funzioni membro poichè membri della classe pila
        // solo le funzioni membro possono accedere alle variabili private
        // PILA e puntatore_pila
        // all'interno della dichiarazione di pila le funzioni membro sono identificate dai prototipi
        // in C++ tutte le funzioni devono avere un prototipo
};
// Inizializza la pila
// Bisogna dire al compilatore la classe a cui appartiene la funzione
// la funzione visualizza è funzione membro della classe pila
// l'operatore "::" chiamato operatore di risoluzione del campo di azione
// specifica che inizializza è nel campo di azione di pila void pila::inizializza()
void pila::inizializza()
{
        puntatore_pila= 0;
}
// Inserimento di un intero nella pila
void pila::push(int i)
{
        if(puntatore_pila==SIZE)
        {
          cout << "Pila piena\n";
          return;
        }
        PILA[puntatore_pila]=i;
        puntatore_pila++;
}
// Estrazione di un intero dalla pila
int pila::pop()
{
        if(puntatore_pila == 0)
        {
          cout << "Pila vuota\n";
          return 0;
        }
        puntatore_pila--;
        return PILA[puntatore_pila];
}
int main()
{
        pila pila1, pila2 ;
        // creazione di due oggetti della classe pila
        // il nome della classe diviene uno specificatore di tipo
        // Ricordate struct?
        // pila1 e pila2 sono oggetti ovvero istanze della classe pila
        // la classe è un'astrazione logica
        // gli oggetti sono reali ovvero esistono in memoria
        pila1.inizializza();
        // quando si deve fare riferimento ad un membro di una classe (dato o funzione che sia)
        // da una parte di codice che non si trova all'interno della classe stessa
        // tale operazione deve essere eseguita sempre in congiunzione con un oggetto di tale
classe
        // per fare questo si usa l'operatore "." chiamato operatore di selezione
        // pila1.puntatore_pila=0; non consentito nel main perchè puntatore_pila è private
```

```
        pila2.inizializza();
        pila1.push(1);
        pila2.push(2);
        pila1.push(3);
        pila2.push(4);
        // cout: canale di output ovvero console di output
        cout << "speriamo che funzioni, " <<"boh?" << endl <<endl;
        cout << pila1.pop() << endl;
        cout << pila1.pop() << endl;
        cout << pila2.pop() << endl;
        cout << pila2.pop() << "\n";
        printf("\n");
        return 0;
}

//OUTPUT
/*risultato
 speriamo che funzioni, boh?
 3
 1
 4
 2
 Press any key to continue
 */
```

1.2.EREDITARIETA'

E' il processo grazie al quale un oggetto acquisisce le proprietà di un altro oggetto,nella fattispecie le variabili e le funzioni di un altro oggetto.

```
Class persona{
        private:
        string nome:
        string cognome;
        public:
        assegnanome();
        visualizzanome();
        assegnacognome();
        visualizzacognome();
        };
```

Uno **studente** è una particolare **persona** che è anche uno **studente**,avrà quindi le stesse variabili e funzioni di persona ed in più avrà variabili e funzioni specifiche.

```
Class studente:public persona{
private:
        string matricola;
public:
        assegnamatricola();
        visualizzamatricola();
};
```

Si parla perciò di oggetto PADRE o BASE (**persona**) e di oggetto FIGLIO o DERIVATO (**studente**). L'oggetto PADRE è detto **SUPERCLASSE**,l'oggetto DERIVATO è detto **SOTTOCLASSE**.
La relazione di ereditarietà è detta relazione "**is a**" (↑).

Esiste un'altra relazione fondamentale nella programmazione orientata agli oggetti.
La relazione di aggregazione o composizione,detta relazione "**has a**" (←).
L'oggetto **ruota** appartiene all'oggetto **bicicletta**.
La **bicicletta** possiede l'oggetto **ruota**.
L'oggetto **ruota** è un membro dell'oggetto **bicicletta**.
La **bicicletta** non è tale se non ha la **ruota**.
La **ruota** senza la superclasse **bicicletta** non ha ragione di esistere.

Esempioutilesulconcettodiereditarietà

```
#include <iostream>
using namespace std;
class edificio
{
        int stanze;
        int piani;
        int superficie;
public:
        void imposta_stanze(int);
        int prendi_stanze();
        void imposta_piani(int);
        int prendi_piani();
        void imposta_superficie(int);
        int prendi_superficie();
};
// casa deriva da edificio
class casa:public edificio
{
        int stanze_da_letto;
        int bagni;
public:
        void imposta_stanze_da_letto(int);
        int prendi_stanze_da_letto();
        void imposta_bagni(int);
        int prendi_bagni();
};
// anche scuola deriva da edificio
class scuola:public edificio
{
        int aule;
        int uffici;
public:
        void imposta_aule(int);
        int prendi_aule();
        void imposta_uffici(int);
        int prendi_uffici();
};
void edificio::imposta_stanze(int num)
{
        stanze=num;
}
void edificio::imposta_piani(int num)
{
        piani=num;
}
void edificio::imposta_superficie(int num)
{
        superficie=num;
}
int edificio::prendi_stanze()
{
        return stanze;
}
int edificio::prendi_piani()
{
        return piani;
}
int edificio::prendi_superficie()
{
        return superficie;
```

```
}
void casa::imposta_stanze_da_letto(int num)
{
	stanze_da_letto=num;
}
void casa::imposta_bagni(int num)
{
	bagni=num;
}
int casa::prendi_stanze_da_letto()
{
	return stanze_da_letto;
}
int casa::prendi_bagni()
{
	return bagni;
}
void scuola::imposta_aule(int num)
{
	aule=num;
}
void scuola::imposta_uffici(int num)
{
	uffici=num;
}
int scuola::prendi_aule()
{
	return aule ;
}
int scuola::prendi_uffici()
{
	return uffici;
}
int main()
{
	casa c;
	scuola s;
	c.imposta_stanze(12);
	c.imposta_piani(3);
	c.imposta_superficie(4500);
	c.imposta_stanze_da_letto(5);
	c.imposta_bagni(3);
	cout<<" la casa ha "<<c.prendi_stanze_da_letto();
	cout<<" camere da letto\n";
	s.imposta_stanze(200);
	s.imposta_aule(180);
	s.imposta_uffici(5);
	s.imposta_superficie(25000);
	cout<<" la scuola ha "<<s.prendi_aule();
	cout<<" aule\n";
	cout<<" la sua superficie vale "<<s.prendi_superficie()<<"\n";
	return 0;
}

//OUTPUT
/* risultato
 la casa ha 5 camere da letto
 la scuola ha 180 aule
 la sua superficie vale 25000
 Press any key to continue
		*/
```

3.POLIMORFISMO

E' il processo grazie al quale è possibile avere funzioni diverse con lo stesso nome. Consiste nel ridefinire una certa funzione cambiandone gli argomenti in numero o tipo. Si dice che le funzioni sono state sovraccaricate (overloading).Il polimorfismo è supportato dal C++ sia al momento della compilazione (compile-time) a mezzo di funzioni ed operatori modificati tramite overloading, sia a runtime per mezzo delle funzioni virtuali.

La programmazione ad oggetti nasce prima di tutto dall'esigenza di gestire programmi di grandi dimensioni
I vantaggi sono numerosi:
Protezione dei dati grazie all'incapsulamento (information hiding).
Maggiore semplicità nella progettazione.
Migliore riutilizzazione del codice.
Migliore manutenzione del codice.
Migliore documentazione.
E' possible, infatti, utilizzare interfacce grafiche che, come i diagrammi di flusso del C, facilitano il lavoro del programmatore. Nel C++ si utilizza
UML.
Il C++ utilizza la Stl, Standard template Library, che è costituita da una serie di routine generiche per la manipolazione dei dati.

Di seguito troveremo due piccoli programmini riguardanti l'overloading di funzioni...ricordiamo che l'overloading è un esempio di polimorfismo

Esempio1

```
#include <iostream>
using namespace std;
// valore_assoluto assume tre significati diversi grazie all'overloading
// il compilatore determina la funzione da richiamare sulla base del tipo di argomento
// il tipo e/o il numero dei parametri deve essere diverso
int valore_assoluto(int);
double valore_assoluto(double);
long valore_assoluto(long);
int main()
{
        cout <<valore_assoluto(-10)<<"\n";
        cout <<valore_assoluto(-11.0)<<"\n";
        cout <<valore_assoluto(-9L)<<"\n";
        return 0;
}
int valore_assoluto(int i)
{
        cout <<"valore_assoluto() per interi\n";
        return i<0 ? -i:i;
}
double valore_assoluto(double d)
{
        cout <<"valore_assoluto() per double\n";
        return d<0.0 ? -d:d;
}
long valore_assoluto(long l)
{
        cout <<"valore_assoluto() per interi long\n";
        return l<0 ? -l:l;
}
/* N.B. torneremo a parlare di overloading di funzioni e di
 possibili
 conseguenti ambiguità, dovute a conversioni automatiche di
 tipo */
```

```
//OUTPUT
/* risultato valore_assoluto() per interi
 10
 valore_assoluto() per double
 11
 valore_assoluto() per interi long
 9
 Press any key to continue
 */
```

Esempio 2

```
#include <iostream>
//#include <cstdio>
//#include <cstring>
using namespace std;
// valore_assoluto assume tre significati diversi grazie all'overloading
// il compilatore determina la funzione da richiamare sulla base del tipo di argomento
// il tipo e/o il numero dei parametri deve essere diverso
void concatena(char *, char *);
void concatena(char *, int);
int main()
{
	char stringa[80];
	strcpy(stringa, "Salve ");
	concatena(stringa, "a tutti voi che siete");
	cout<<stringa<<"\n";
	concatena(stringa, 100);
	cout<<stringa<<"\n";
	return 0;
}
//concatena due stringhe
void concatena(char *stringa1, char *stringa2)
{
	strcat(stringa1,stringa2);
}
//concatena una stringa con un intero convertito in stringa
void concatena(char *stringa1, int i)
{
	char temp[80];
	sprintf(temp," %d",i);
	strcat(stringa1,temp);
}

//OUTPUT
/* risultato
 Salve a tutti voi che siete
 Salve a tutti voi che siete 100
 Press any key to continue
 */
```

CLASSI

Istanziare vuol dire allocare memoria.
L'oggetto è l'istanza di una classe.
La classe,come la struct,è un'astrazione. E' un nuovo tipo definito dal programmatore.
In C++ si lavora con le classi.
Le variabili e le funzioni all'interno di una classe possono essere di 3 tipi:

private,**public**,**protected**.
Le variabili **private** sono visibili solo dalle funzioni che appartengono alla stessa classe e nascoste alle altre(information hide). E' un modo di fare incapsulamento. Private è **pleonastico** e quindi si può omettere. Tutte le variabili e le funzioni dichiarate dopo **public**: possono essere utilizzate da tutte le altre funzioni del programma. Variabili e funzioni dichiarate all'interno del programma si dicono variabili membro e funzioni membro. Solo le funzioni membro hanno accesso ai membri privati della classe in cui sono dichiarate. L'utilizzo di variabili private garantisce una PROGRAMMAZIONE ROBUSTA.
Una variabile **protected** rimane privata se non per gli oggetti derivati dall'oggetto base. E' accessibile solo dalle classi derivate,inaccessibili dall'esterno.
Per accedere ai singoli membri dell'oggetto si usa l'operatore "**.**" in maniera simile alle struct.L'operatore punto è l'unico che rispetta il principio di **località.**
E' necessario tenere sempre ben chiara l'occupazione di memoria di un oggetto. Essa coincide con l'occupazione di memoria delle sue variabili e solo di esse. Le funzioni non occupano spazio in memoria.

Nota: Il principio di località spaziale e temporale afferma che, se la CPU sta eseguendo una data istruzione, vuol dire che con molta probabilità le prossime istruzioni da eseguire saranno ubicate nelle vicinanze di quella in corso. In altre parole,se all'istante t=t0 hai avuto bisogno dell'indirizzo logico α è molto probabile che all'istante t=t0 +1 abbia bisogno dell'indirizzo logico α+1.

FUNZIONI INLINE/OFFLINE

Una funzione dichiarata e definita all'interno di una classe si dice **INLINE.**
Le funzioni **OFFLINE** invece sono prototipi passati nella classe la cui definizione è fuori della classe,in genere dopo la stessa.
Void persona**::**acquisisci_nome(); //"**::**"è l'operatore risoluzione del campo di azione.
Class persona p; //dichiaro una classe persona di nome p;
p.visualizza(); //Accedo alle funzioni come accedo al campo di un dato strutturato

Esempio di funzione inline

Una funzione inline non viene effettivamente richiamata ma il suo codice viene espanso nel punto in cui dovrebbe essere richiamata, per fare ciò è necessario far precedere alla sua definizione la parola chiave inline.

```
#include <iostream>
using namespace std;
/*inline int max(int a, int b)
 {
 return a>b ? a:b;
 }
 int main()
 {
 cout <<max(10,20) <<endl;
 cout <<"" << max(99,88)<<endl;
 return 0;
 }*/

/* equivalente al seguente programma */
int main()
{
        cout <<(10>20 ? 10:20)<<endl;
        cout <<"" <<(99>88? 99:88)<<endl;
        return 0;
}
```

```
/* risultato
 20
 99
 Press any key to continue
            */
```

N.B. : esiste un secondo metodo di utilizzo di funzioni inline: sono quelle i cui corpi sono direttamente scritti nella classe.
Questo secondo metodo è consigliato, sempre che il codice relativo sia breve!!!
Il metodo è anche ulteriormente consigliato per costruttori e distruttori.

ESERCIZIO 1

```
#include <iostream> //non è un file perchè manca il .h,codice dopo la standardizzazione ISO/ANSI
using namespace std; //per rendere compatibile con le vecchie versioni del c
class studente //dichiaro una classe.
{
private: //variabili private
        int a; //variabile membro
public:
        void inline assegna()
        {
          a=5;
        }
        void inline visualizza() {cout<<a<<"\n\n";};
}/*s*/;
int main(){
        //class studente s;
        studente s;
        //ISTANZIO L'OGGETTO S AD ESSERE UN OGGETTO DELLA CLASSE STUDENTE
        //s; // se lo dichiaro tra gli oggetti della classe
        //s.a=5; //mi darebbe errore perchè non si può accedere ad una variabile privata con una funzione NON membro
        s.assegna();
        //cout<<s.a //stessa cosa, a è privata
        s.visualizza();      //con una funzione pubblica membro posso accedere alla variabile membro della stessa classe
        system("PAUSE");
        return 0;
}

//OUTPUT 5
```

ESERCIZIO 2

```
#include <iostream>
using namespace std;
#define SIZE 100
//creazione della classe stack
class stack{
private: //è sottointeso ma lo scrivo comunque
        int stck[SIZE];
        int tos;
public:
        inline void init(){//FUNZIONE INLINE.Per certi compilatori inline è sottointeso
          tos=0;
        }
        void push(int i);
        int pop();
}; //nessun oggetto dichiarato nella classe
```

```
void stack::push(int i){ //offline
        if (tos==SIZE)
        {
          cout<<"stack esaurito.\n";
          return; //non servirebbe
        }
        stck[tos]=i;
        tos++;
}

int stack::pop(){
        if (tos==0)
        {
          cout<<"stack vuoto.\n";
          return 0; //serve perchè la funzione restituisce un intero
        }
        tos--;
        return stck[tos];
};

/***     MAIN       ***/

int main(){
        stack stack1,stack2; //creo 2 oggetti della classe stack
        stack1.init(); //inizializza tos dell'oggetto stack1 a 0
        stack2.init(); //inizializza tos dell'oggetto stack2 a 0
        stack1.push(1); //inserisce 1 nello stack[tos] e incrementa tos
        stack2.push(2); //inserisce 2 nello stack[tos] e incrementa tos
        stack1.push(3); //inserisce 3 nello stack[tos] e incrementa tos
        stack2.push(4); //inserisce 4 nello stack[tos] e incremente tos
        cout<<stack1.pop()<<" "; //estrae il primo elemento che è 3.Il primo è l'ultimo elemento
perchè è una PILA
        cout<<stack2.pop()<<" "; //estrae il secondo che è 1
        cout<<stack1.pop()<<" "; //estrae il primo di stack 2 che è 4
        cout<<stack2.pop()<<" "; //estrae il secondo che è il primo inserito in stack 2 xche è 2
        system("PAUSE");
        return 0;
}


//OUTPUT 3 4 1 2
```

ESERCIZIO 3

pila dinamica

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

class pila{
private:
        int dim;
        int *V;
        int punt_t;
public:
        pila(int A) //costruttore
        {
          dim=A;
          punt_t=0;
          cout<<"\n\n costruttore \n\n";
          V=new int[dim];
        }
```

```
        void push( );
        void pop( );
        void punt( ){cout<<"\n\t punt_t= "<<punt_t;}//inline automatico
        void visualizza( );

        ~pila( ) //distruttore
        {
          cout<<"\n distruttore chiamato \n\n";
          delete []V;
        }
};
void pila::visualizza( ){
        int i;
        cout<<"\n";
        if(punt_t ==0)
          cout<<"\n\n\t Pila Vuota \n\n";
        else
          for(i=0; i<punt_t;i++)
                  cout<<V[i]<<"\t";
}

void pila::push( )
{
        if(punt_t<dim)
        {
          cout<<"\n Inserire un valore nella pila: ";
          cin>> V[punt_t];
          punt_t++;
        }
        else
          cout<<"\n\n\t pila piena\n\n";
}
void pila::pop( )
{
        if(punt_t>0)
        {
          punt_t--;
          cout<<"Estrazione dell'elemento"<<V[punt_t];
        }
        else
          cout<<"\n\n\t pila vuota \n\n";
}

int main( )
{
        int n;
        cout<<"\n Inserire la grandezza del vettore\n";
        cin>>n;
        pila p(n);//oggetto
        int scelta=-1;
        while (scelta !=0)//Menù
        {
          cout<<"\n 1 push \n 2 pop \n 3 visualizza \n 0 esci\n";
          cin>> scelta;
          if (scelta==1)
          {
                  p.push( );//inserimento dei valori
                  p.punt( ); // visualizza l'indice del vettore
          }
          else
                  if (scelta == 2)
                  {
                          p.pop( );//estrazione dell'elemento
```

```
                    p.punt( );
            }
            else
                    if (scelta == 3)
                            p.visualizza();
}
return 0;
}
```

ESERCIZIO 4

pila statica

```
#include<iostream>
using namespace std;
#define N 5

class pila{
private:
	int V[N];
	int punt_t;
public:
	pila() //costruttore
	{
	  punt_t=0;
	  cout<<"\n\n Il costruttore ha inizializzato punt_t= "<<punt_t<<"\n\n";
	}
	void push();
	void pop();
	void punt(){ cout<<"\n\t punt_t="<<punt_t;};
	~pila() {cout<<"\ndistruttore chiamato\n";}//distruttore
};

void pila::push(){
	if(punt_t<N)
	{
	  cout<<"\n inserire un valore nella pila ";
	  cin>>V[punt_t];
	  punt_t++;
	}
	else
	  cout<<"\n\n\t pila piena\n\n";
}

void pila::pop(){
	if (punt_t>0)
	{
	  punt_t--;
	  cout<<"Estrazione dell'elemento "<<V[punt_t];
	}
	else
	  cout<<"\n\n\t pila vuota \n\n";
}

int main(){
	pila p; //oggetto
	int scelta=-1;
	while(scelta!=0) //per fare la cosa automaticamente
	{
	  cout<< "\n 1 push \n 2 pop \n 0 esci \n";
	  cin>>scelta;
	  if (scelta==1)
	  {
		p.push(); //per inserire i valori
		p.punt(); //visualizza l'indice del vettore
	  }
	  else
		if (scelta==2)
		{
			p.pop(); //per estrarre
			p.punt();
		}
	}
```

```
        system("PAUSE");
        return 0;
        }
```

ESERCIZIO 5:
calcolo di modulo e fase di un numero complesso

```
#include <iostream>
#include <math.h>
#include <conio.h>
using namespace std;

class NC
{
private:
        float RE;
        float IM;
        float Fase;
        float Modulo;
public:
        friend float fase(class NC &);
        friend float modulo(class NC &);
        void assegna (float A,float B) {Fase=B; Modulo=A;}
        void visualizza ( ) {cout<<"\n"<<RE<<"+j"<<IM<<"\n";}
        void visualizza1 ( ) {cout<<"\nModulo "<<Modulo<<" Fase in radianti "<<Fase<<" Fase in
gradi "<<((Fase*180)/3.14)<<"\n";}
        NC( )
        {
          cout<<"\n parte reale: ";
          cin>>RE;
          cout<<"parte immaginaria: ";
          cin>>IM;
        }
};

int main ( )
{
        float M,F;
        do
        {
          cout<<"Immettere numero complesso C1: ";
          NC C1;
          M=modulo(C1);
          F=fase (C1);
          C1.assegna(M,F);
          C1.visualizza( );
          C1.visualizza1( );
          cout<<"\nimmettere il numero complesso C2: ";
          NC C2;
          M=modulo(C2);
          F=fase(C2);
          C2.assegna(M,F);
          C2.visualizza( );
          C2.visualizza1( );
          cout<<"premere n per uscire, altrimenti premere un tasto\n\n";
        }
        while (getch()!='n');
        return 0;
}

float fase( class NC & A)
{
        return (float) (atan2((float)A.IM, (float)A.RE));//double atan2(double y,duoble x); y/x
}
```

```
float modulo (class NC & A)
{
        return (float) (sqrt(pow(A.RE,2)+pow(A.IM,2)));
}
```

Ecco un'altra soluzione al problema del calcolo del modulo di un numero complesso attraverso l'utilizzo delle classi

```
#include <iostream>
#include <math.h>
#define pi 3.14159
using namespace std;
class n_c {
private: float p_r; float p_i; public:
   n_c(); //costruttore non parametrizzato;
   float modulo;
   void stampa();
   void stampa(float);
   void inserisci(float , float);
   void inserisci();
   float calcola_modulo();
   friend float arg(n_c);
};
n_c::n_c()
{
   cout <<"inizializzo" << endl <<endl;
   p_r=0.0;
   p_i=0.0;
}
void n_c::stampa()
{
   cout<<"(" << p_r << " , " << p_i << ")"<<endl<<endl;
}
void n_c::stampa(float)
{
   cout<<"il modulo di (" << p_r << " , " << p_i << ") vale " <<
   modulo << endl<< endl;
}
void n_c::inserisci(float i, float j)
{
   p_r = i;
   p_i = j;
}
void n_c::inserisci()
```

```
{
   cout <<"inserisci parte reale e parte immaginaria" << endl
   <<endl;
   cin >> p_r >> p_i ;
}
```

```
float n_c::calcola_modulo()
{ modulo=(float)sqrt(p_r*p_r+p_i*p_i); return modulo;
}
//attenzione agli angoli
float arg(n_c copia)
{
   return ((atan(copia.p_i/copia.p_r))/pi)*180;
}
int main()
{
   n_c C1,C2;
   cout <<"valore iniziale dei numeri complessi" <<endl <<endl; C1.inserisci(1,1);
   cout <<"primo numero complesso"<< endl; C1.stampa();
C1.stampa(C1.calcola_modulo()); C2.inserisci();
   cout << endl << "secondo numero complesso"<< endl; C2.stampa();
   C2.stampa(C2.calcola_modulo());
   cout << endl << "mi ripeto vale " << C2.modulo << endl <<
   endl;
   cout <<"la fase dei due numeri vale in gradi " << endl << endl
   << arg(C1) << endl <<endl << arg(C2) <<endl <<endl;
   return 0;
}
```

OUTPUT

```
/* risultato
inizializzo
inizializzo
valore iniziale dei numeri complessi
primo numero complesso
(1 , 1)
il modulo di (1 , 1) vale 1.41421
inserisci parte reale e parte immagina
1
0
```

```
secondo numero complesso
(1 , 0)
il modulo di (1 , 0) vale 1
mi ripeto vale 1
la fase dei due numeri vale in gradi
45
0
Press any key to continue
*/
```

COSTRUTTORI E DISTRUTTORI

Il costruttore ha lo scopo di inizializzare le variabili membro. Il distruttore ha funzione opposta,(like allocare e disallocare memoria,aprire e chiudere un file). Costruttori e distruttori sono **chiamati automaticamente** quando viene dichiarato un oggetto Sono di **default**. Sono funzioni che non restituiscono nessun valore ed hanno lo stesso nome della classe. Il distruttore è preceduto dalla tilde ~. Se devo passare un oggetto ad una funzione,l'oggetto deve essere costruito ma non devo passare il costruttore.
Come già detto,il costruttore serve solo per inizializzare. Non sono chiamati per passare oggetti temporanei. Creo l'oggetto padre,poi l'oggetto figlio. Le variabili di ogni classe sono inizializzate con i propri costruttori,definiti inline nella sezione public. Le variabili ereditate inizializzate con il costruttore del padre,quelle invece che specializzano l'oggetto figlio devono essere inizializzate a mezzo dei costruttori nell'oggetto figlio. La distruzione avviene in senso opposto alla costruzione, secondo una gerarchia :
Costruttore padre,costruttore figlio,distruttore figlio, distruttore padre.
In altre parole,mentre il costruttore del padre viene eseguito prima di quello del figlio,il distruttore del padre viene eseguito dopo quello del figlio.

Esempio di uso dei costruttori e distruttori

```
#include <iostream>
using namespace std;
#define SIZE 100
class pila {
    int PILA[SIZE];
int puntatore_pila;
public:
pila();//costruttore
/* una funzione costruttore è una particolare funzione membro di una classe che porta lo
 stesso nome della classe, le funzioni costruttore non possono restituire valori
 pertanto non si deve specificare il tipo restituito scopo delle funzioni costruttore è
eseguire le
 inizializzazioni*/
~pila();//distruttore
/*la funzione distruttore ha lo stesso nome del costruttore ma preceduto da "~" "126"
 scopo della funzione distruttore è deallocare la memoria precedentemente allocata
dall'oggetto*/
void push(int);
int pop();
};
//funzione costruttore
pila::pila()
{
```

```
    puntatore_pila=0;
    cout<<"pila inizializzata\n";
}
//funzione distruttore
pila::~pila()
{
    cout<<"pila distrutta\n";
}
// Inserimento di un intero nella pila
void pila::push(int i)
{
    if(puntatore_pila==SIZE) {
        cout << "Pila piena\n";
        return;
        } PILA[puntatore_pila]=i; puntatore_pila++;
        }
        // Estrazione di un intero dalla pila
int pila::pop()
        {
            if(puntatore_pila == 0) {
                cout << "Pila vuota\n";
                return 0;
            }
            puntatore_pila--;
            return PILA[puntatore_pila];
        }
        int main()
        {
            pila pila1, pila2 ; pila1.push(1); pila2.push(2); pila1.push(3);
            pila2.push(4);
            cout << pila1.pop() << " ";
            cout << pila1.pop() << " "; cout << pila2.pop() << " "; cout << pila2.pop() << "\n";
return 0;
        }
```

OUTPUT

```
/*
risultato:
pila inizializzata
pila inizializzata
3 1 4 2
pila distrutta
pila distrutta
Press any key to continue */
```

ESERCIZIO 6

```
//qualcosa con le stringhe e concatenamento stringhe
//carico nel vettore allocato dinamicamente dei numeri e li
ordino
#include <iostream>
using namespace std;
#define SIZE 10

class frase{
private:
    char fr1[SIZE];
    char fr2[SIZE];
    char ff[2*SIZE];
```

```
public:
    static int conteggio;
    friend void concat(frase & A);
    void visualizza(){
        int i;
        printf("\n\n frase %s        \n",fr1);
        printf("\n\n frase %s        \n",fr2); printf("\n\n concatenamento  ");
for(i=0;i<(2*SIZE);i++){
            printf("\n ");
            printf("%d        %c",i,ff[i]);
        }
        /* printf(" concatenamento %s ",ff); ne visualizzerebbe
         solo una perchè copia anche il terminatore di
         stringa*/
    }
    void fvisualizza(){
        cout<<"\n\n con la funzione "<<ff;
    }
    void fconcat(){
        strcpy(ff,fr1);
        strcat(ff,fr2);
    }
    void contegg(){
        cout<<"\n\t conteggio ="<<conteggio<<"\n";
    }
    frase(){
        strcpy(fr1,"          ");
        strcpy(fr2,"                    "); strcpy(ff,"     "); conteggio++;
    }
    void inserisci(){
        printf("\n inserire la prima stringa ");
        scanf(" %s",fr1);
        printf("\n inserire la seconda stringa ");
        scanf(" %s",fr2);
    }
    ~frase(){
        conteggio--;
    }
};

class ordinamento{
private:
    int *V;
    int N;
public:
    void inserisci(){
        int i;
        cout<<"Inserisci elementi vettore \n";
        for (i=0;i<N;i++)
```

```
            cin>>V[i];
    }
    ordinamento(){ //costruttore
        printf("\n Inserire la dimensione del vettore: ");
        cin>>N;
        V=new int [N];
    }
    ~ordinamento(){ //distruttore
        delete []V;
    }

    void ordina();
    void visualizza(int A){
        int i;
        if (A==0)
            printf( "\n vettore inserito");

        else


            printf("\n vettore ordinato");

        for(i=0;i<N;i++)
            printf("\n V[%d]=%d",i,V[i]);
    }
};

void ordinamento::ordina(){ //con il selection sort
    int i,imin,j,temp;
    i=0;
    j=i+1;
    for(i=0;i<N-1;i++){
        imin=i;
        for(j=i+1;j<N;j++)
            if (V[j]<V[imin])
                imin=j;
        temp=V[i]; V[i]=V[imin]; V[imin]=temp;
    }
}

int frase::conteggio;    //la variabile static è inizializzata a 0 automaticamente

int main(){ frase f; ordinamento *VE; f.contegg(); f.inserisci(); concat(f); f.visualizza();

    f.fconcat();
    f.fvisualizza();

    ordinamento VETT; VE=&VETT;
    cout<<"\nchiamata a run-time\n";
    VE->inserisci(); //chiamata a runtime
    VE->visualizza(0); VE->ordina();
    VE->visualizza(1);
    //chiamate a compile time
    cout<<"\nchiamata a compile-time\n"; VETT.inserisci();
    VETT.visualizza(0); VETT.ordina(); VETT.visualizza(1); system("PAUSE"); return 0;
}
```

```
void concat(frase & A){
   int i,j;
   for (i=0;i<SIZE;i++){
      A.ff[i]=A.fr1[i];
   }
   for (j=i,i=0;i<SIZE;i++,j++){ A.ff[j]=A.fr2[i];
   }
}
```

FRIEND

Una funzione si dice **friend** (letteralmente “amica”) di una classe,diversa da quella di appartenenza,se può accedere a tutte le sue variabili private. La dichiarazione di una funzione friend è molto semplice: basta inserire il prototipo della funzione nella definizione della classe, preceduto dalla parola chiave “friend” (non importa se nella sezione protetta o pubblica).Sono dichiarate quindi nella classe ma non appartengono alla classe. Le funzioni non dichiarate nella classe sono dette invece **di servizio**. Siccome è esterna può accedere agli oggetti della classe solo se l'oggetto interessato le è trasmesso come argomento e poiché non è una funzione membro va richiamata senza utilizzare l’operatore ”**.**”. Anche le classi possono essere friend. In questo modo la classe friend e tutte le sue funzioni membro avranno accesso ai membri privati e protected definiti all’interno dell’altra classe, ma non ne ereditano le caratteristiche. Si “prototipizzano” nella classe in cui ci sono le variabili private cui devono accedere ma la loro definizione è fatta all'esterno,prima o dopo il main,come una normale funzione C. Se passo la classe a queste funzioni,x accedere alle variabili private devo utilizzare l'operatore punto “**.**” . Le funzioni membro ricevono automaticamente un

puntatore **this** e quindi possono accedere alle variabili private. Le funzioni friend possono accedere alle variabili della classe solo se tra gli argomenti c'è la classe.

Esempio utile per l'uso di funzioni friend

```
#include <iostream>
using namespace std;
class mia_classe { int a,b;
public:
friend int somma(mia_classe);
void imposta_ab(int,int);
};
void mia_classe:: imposta_ab(int i, int j)
{
   a=i;
   b=j;
}
// somma() non è funzione membro di alcuna classe
int somma(mia_classe x)
{
   // dal momento che somma è friend di mia_classe ha accesso ad a ed a b
   return x.a+x.b;
}
int main()
{
   mia_classe classe1;
   classe1.imposta_ab(3,4);
   cout << somma(classe1) <<endl;
   return 0;
}
```

ESERCIZIO 7
`funzione friend,passaggio di classe con reference,classe figlio`

```
#include <iostream>
using namespace std;
class persona{ private:
char nome[10];
char cognome[20];
int voto;
public:
void v_nome(); void v_cognome(); void v_voto();
friend void mod_vot(persona&);
persona() //costruttore
{
   cout<<"\n costruisci \n"; strcpy(nome, "francesco"); strcpy(cognome,"rossi"); voto=10;
}
~persona(){ cout<<" \n distrugge\n";}
};
```

```
void persona::v_cognome(){cout<<"\nnome: "<<nome<<"\n";} void
persona::v_nome(){cout<<"\ncognome: "<<cognome<<"\n";} void
persona::v_voto(){cout<<"\nvoto: "<<voto<<"\n";}

class studente:public persona
{
private:
   int matricola;
public:
   studente(int mat) //costruttore
   {
      matricola=mat;
   }
   studente()
   {
      matricola=0;
   }
   void caricamat(){
      matricola=111111;}
   void v_matricola();
};

void studente::v_matricola()
{cout<<"\nmatricola :"<<matricola<<" ";}
void mod_vot(persona &A){
   cout<<"\n Inserire il voto,funzione friend \n";
   cin>>A.voto;
}
int main(){
   persona p; //costruttore
   cout<<"\n Studente P: "; p.v_nome(); p.v_cognome(); p.v_voto();
   studente s; //costruttore chiamata cout<<"\n Studente S: ";
   s.v_cognome(); ///posso usarlo perchè s è studente ed è figlio di persona
   s.v_nome(); s.v_voto(); s.caricamat(); s.v_matricola();
   mod_vot(p); //la funzione friend si chiama come una funzione normale
   cout<<"\n voto di P: ";
   p.v_voto();
   mod_vot(s); //ha ereditato anche la funzion friend cout<<"\n voto di S: ";
   s.v_voto(); system("PAUSE"); return 0;
}
```

Ulteriore esempio per le classi friend

```
// in questo caso la classe friend e tutte le sue funzioni
// membro avranno accesso ai membri privati definiti all'interno della classe
// in questo caso la classe Min ha accesso alle variabili a e b nella classe TwoValues
#include <iostream>
using namespace std;
class TwoValues
{
   int a;
   int b;
public:
   TwoValues(int i, int j) { a = i; b = j; }
   friend class Min;
```

```
};
class Min
{
public:
   int min(TwoValues x);
};
int Min::min(TwoValues x)
{return (x.a < x.b ? x.a : x.b);}
int main()
{
   TwoValues ob(10,20);
   Min m;
   cout << m.min(ob)<<endl;
   return 0;
}
```

OPERATORE THIS

Quando viene richiamata una funzione membro,le viene automaticamente passato un argomento implicito costituito da un puntatore all'oggetto chiamante. Questo puntatore è chiamato **this**. Ogni oggetto può quindi accedere al proprio indirizzo tramite il puntatore this. Il puntatore this non fa parte dell'oggetto. Può essere usato in maniera implicita(senza essere richiamato) oppure può essere esplicitato. E' a mezzo dell'operatore this che le funzioni possono automaticamente accedere alla loro variabili private.

Esempio

```
#include <iostream>
using namespace std;
class pwr {
double b;
int e; double val; public:
pwr(double base, int exp);
double get_pwr() {return val;}
};
pwr::pwr(double base, int exp)
{
   b = base; e = exp; val = 1;
   if(exp==0) return;
   for( ; exp>0; exp--) val = val*b;
}
int main()
{
   pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
   cout << x.get_pwr() << " ";
   cout << y.get_pwr() << " "; cout << z.get_pwr() << "\n";
   return 0;
}
/*
 scrittura alternativa di pwr() con il puntatore this:
 pwr::pwr(double base, int exp)
 {
 this->b = base; this->e = exp; this->val = 1;
 if(exp==0) return;
 for( ; exp>0; exp--)
 this->val = this->val * this->b;
 }
```

*/

MEMORIA DINAMICA

Una variabile a dichiarata
int a;
è memorizzata nello stack.
Per allocare dinamicamente una variabile nella memoria **heap** si usa l'operatore **new**.
Variabile= new tipo;
Per disallocare variabili allocate dinamicamente si usa l'operatore **delete**.

C	C++

Int *p;
p=malloc(sizeof());
free(p);
	Int *p;
	p=new int;
	delete p;

La funzione new(standard) se non va a buon fine genera una **eccezione,**cioè una routine che cerca di risolvere il problema.

ALLOCAZIONE MATRICI

La logica che si adotta è la stessa del C. Creo il vettore colonna di puntatori e poi le righe.
Per disallocare,in maniera inversa,prima le righe e poi le colonne.

Int **A; //R e C sono righe e colonne e devono essere inizializzate
//Nel costruttore:

```
A=new int*[R];
for (i=0;i<R;i++)
A[i]=new int [C];
//Nel distruttore:
for(i=0;i<R;i++)
delete[] A[i];
delete[]A;
```

Le parentesi[] prima dei vettori A servono proprio a far capire al compilatore che sono vettori quindi a far disallocare tutta la memoria occupata dal vettore. Ovviamente il 2° delete è un vettore di puntatori.

ESERCIZIO 8
ingrandire immagine e contare numero di nero nell'immagine

```
#include <iostream>
using namespace std;
class immagine
{
private:
   int R;
   int C;
   unsigned char** IMM;
public:
   int colonna(){return C;}
   int riga(){return R;}
   unsigned char** imm(){return IMM;}
   void conta ()
   {
      int i,j,cont=0;
      for(i=0;i<R;i++)
         for(j=0;j<C;j++)
         {
            if(IMM[i][j]=='0')
               cont ++;
         }
      cout<<cont;
   }
   void visualizza()
   {
      int i,j; cout<<"\n\n"; for(i=0;i<R;i++)
      {
         for(j=0;j<C;j++)
            printf("%3d",IMM[i][j]);
         cout<<"\n";
      }
   }
   void dati()//è nella classe e non ha bisogno dei parametri
   {
      int i,j;
      for(i=0;i<R;i++)
         for(j=0;j<C;j++)
         {
```

```
            printf("\n Inserire l'elemento IMM[%d][%d]=",i,j);
        }


          cin>>IMM[i][j];
          }


          friend void nero(immagine &);//passo per reference
  immagine(int x,int y)//costruttore
    {
        int i; R=x; C=y;
        cout<<"\n costruttore\n";
        IMM=new unsigned char *[R];
        for (i=0;i<R;i++)
            IMM[i]=new unsigned char[C];
    }
          immagine (){};
          ~immagine()
    {
        int i;
        printf("\n\n distruttore");
        for(i=0;i<R;i++)
            delete[]IMM[i];

    }
          };

          class zoom:public immagine{
          private:
              int fattore;
              unsigned char **ZOOM;
              int RR;
              int CC;
          public:
              int colonnaz(){return CC;}
              int rigaz(){return RR;}
              unsigned char** immz(){return ZOOM;}
              void visualizza()
              {
                  int i,j; cout<<"\n\n"; for(i=0;i<RR;i++)
                  {
                      for(j=0;j<CC;j++)
                          printf("%3d",ZOOM[i][j]);
                      cout<<"\n";
                  }
              }
              zoom(int A,int B,int C):immagine(A,B){//costruttore che passa anche i valori
della classe base
                  fattore=C;
                  cout<<"\n costruttore zoom \n";
                  int i=0,j=0,ii=0,jj=0,x=0,y=0;
                  unsigned char **M=imm();//non ha senso...
                  RR=fattore*A; CC=fattore*B;

                  ZOOM=new unsigned char* [RR];
                  for(i=0;i<RR;i++)
```

```
            ZOOM[i]=new unsigned char [CC];
        }
        ~zoom(){
            int i;
            printf("\n\nzoom distrutto \n\n");
            for(i=0;i<RR;i++)
                delete[]ZOOM;
        }
    };

    int main()
{
    int r,c,f,ii=0,jj=0,x=0,y=0,i=0,j=0;
    unsigned char **Zu,**In;
    cout<<"Inserire il numero di righe e colonne\n";
    cin>>r>>c;

    immagine I(r,c); I.dati();









    I.visualizza();
    nero(I);
    cout<<"Inserire il fattore di ingrandimento\n";
    cin>>f;
    r=I.riga(); c=I.colonna(); zoom z(r,c,f); r=z.rigaz(); c=z.colonnaz();

    Zu=z.immz(); In=I.imm();
    //ingrandimento
    for(ii=0,x=0;ii<r;ii=ii+f,x++)//per saltare di f in f
    {for(jj=0,y=0;jj<c;jj=jj+f,y++)
    {
        for(i=ii;i<(ii+f);i++)
            for(j=jj;j<(jj+f);j++)
                Zu[i][j]=In[x][y];
    }
    }
    z.visualizza();
    system("PAUSE");
    return 0;
}

    void nero(immagine & M)
{
    int i,j,cont=0;
    for (i=0;i<M.R;i++)
        for(j=0;j<M.C;j++)
        {
            if(M.IMM[i][j]==0)//per far funzionare if
                cont++;
                }
    printf("\n\n Il valore di nero = %d\n\n",cont);
```

}

PUNTATORE A FUNZIONE E OPERATORE &

In C il passaggio di argomenti alle funzioni avveniva per valore o per copia di indirizzo (puntatore).

In C++ si utilizza il **passaggio per indirizzo** che chiameremo per riferimento,**reference** e serve ad evitare la dereferenziazione,cioè l'operazione di risoluzione del riferimento.

C	C++

Void scambia (int *p,int *q);
scambia(&a,&b); //chiamata funzione
/*In p finisce l'indirizzo di a ed in q l'indirizzo di b.*/
	Void scambia(int& p,int& q);
	scambia(a,b); //chiamata funzione.
	/*a e b sono 2 nuove variabili né di tipo int né di tipo int*.Sono di tipo **int&** ,variabili reference*/

Esempio di puntatore a funzioni in C

```
#include <stdio.h>
int controlla_num(float);
int inverti_num(int (*)(float),float * );
int main()
{
   float x;
   int (*p)(float)=controlla_num; printf ("inserisci x\n"); scanf("%f",&x);
   if ((inverti_num(p,&x))==0)
      printf("l'inverso vale %f\n", x);
   return 0;
}
int inverti_num(int(*funz)(float X),float *X)
{
   if ((funz)(*X) ==0) // programmatori esperti ed un po' anticonformisti
      //if ((*funz)(*X) ==0) ... dei veri accademici della programmazione !!!
   {*X=1/(*X);
      return 0;}
   else return 1;
}
int controlla_num(float X)
{
   if (X==0)
   {printf("funzione inverso non definita\n");
      return 1;}
   else return 0;
}
```

Esempio di puntatore a funzioni e reference in C++

```
include <iostream>
using namespace std;
int controlla_num(float &);
int inverti_num(int (*)(float &),float &);
int main()
{
   float x;
   int (*p)(float &)=controlla_num;
   cout <<"inserisci x"<<endl;
   cin >> x;
   if ((inverti_num(p,x))==0)
      cout <<"l'inverso vale "<< x<<endl;
   return 0;
}
int inverti_num(int(*funz)(float &X),float &X)
{
   if ((*funz)(X) ==0) // vedi esempio precedente
   {X=1/(X);
      return 0;}
   else return 1;
}
int controlla_num(float &X)
{
   if (X==0)
   {cout<<"funzione inverso non definita"<<endl;
      return 1;}
   else return 0;
}
```

Esempio di puntatore a funzioni e reference (non troppi) in C++

```
#include <iostream>
using namespace std;
int controlla_num(float);
int inverti_num(int (*)(float),float &);
int main()
{
   float x;
   int (*p)(float)=controlla_num;
   cout <<"inserisci x"<<endl;
   cin >> x;
   if ((inverti_num(p,x))==0)
      cout <<"l'inverso vale "<< x<<endl;
   return 0;
}
int inverti_num(int(*funz)(float X),float &X)
{ if ((*funz)(X) ==0)
```

```
{X=1/(X);
   return 0;}
   else return 1; }
int controlla_num(float X)
{
if (X==0)
{
   cout<<"funzione inverso non definita"<<endl;
   return 1;}else return 0;
}
```

BINDING

Per binding si intende una connessione tra la chiamata di una funzione ed il codice che la implementa.
Si definisce **binding statico** (o precoce o anticipato) quando il compilatore genera il codice per chiamare un dato metodo ogni volta che quel metodo viene applicato ad un dato oggetto. Si utilizza l'operatore punto "**.**" per accedere alle variabili.
Si definisce **binding dinamico** (o tardivo o ritardato) quando il compilatore non genera una volta per tutte, all'atto della compilazione, il codice per l'assegnazione dei valori delle variabili in funzione delle chiamate dei metodi, o il codice per calcolare quale metodo chiamare in funzione delle informazioni provenienti dall'oggetto,ma genera un codice di volta in volta durante l'esecuzione del programma ed all'atto della chiamata della funzione(run-time).Si usa il passaggio per puntatore.

FUNZIONI VIRTUALI

Un presupposto del polimorfismo sono le funzioni virtuali. E' una funzione membro dichiarata come **virtual** in una classe base e ridefinita in una classe derivata. La classe derivata ridefinisce la funzione virtuale secondo le sue esigenze. Si dice infatti **un interfaccia per più metodi**. Le funzioni virtuali servono per effettuare il binding dinamico. Il limite è che per le variabili puntatore di funzione virtuale l'aritmetica dei puntatori è quella dell'oggetto padre.
OVERLOADING di funzioni → a compile-time
OVERRIDDING di funzioni → a run-time.(ha senso solo per funzioni ereditate)

La funzione figlio può avere stesso nome e stessi parametri della funzione padre ma deve avere corpo diverso.

TIPO RESTITUITO VIRTUAL NOME () {}

void virtual stampa();

Nella gerarchia creata dalle varie classi ereditate,scelgo dove iniziare l'overridding.Il tipo virtual si mette solo alla prima classe. Per le classi ereditate non serve più ed è possibile ridefinire le funzioni. Se uso i puntatori,basta cambiare l'assegnazione al puntatore e la chiamata alla funzione virtuale rimane la stessa ma assume ogni volta valore diverso.
Un puntatore alla classe base può puntare a qualsiasi classe derivata dalla base quando un puntatore punta ad un oggetto derivato che contiene una funzione virtuale il C++ determina quale versione di tale funzione richiamare sulla base del tipo di oggetto puntato dal puntatore e questo a run-time mentre l'oggetto può cambiare

FUNZIONI VIRTUAL PURE

Definisco una funzione virtual pure una funzione che non ha corpo, essa non viene definita all'interno della classe base.
Void virtual stampa()=0;
Inizializzo il puntatore alla funzione a NULL. Negli oggetti figli la DEVO ridefinire per non avere ERRORE. Serve per una **programmazione robusta**,per ricordare che devo definire la funzione per ogni classe.
Quando in una classe è definita una funzione virtual pure la classe si chiama ASTRATTA.

Una **CLASSE ASTRATTA** ha
1.Una funzione virtual pure
2.Non vi è possibile istanziare oggetti.
3.Si istanzia un puntatore o un reference.
4.E' come un prototipo e solo gli oggetti figli possono essere manipolati.

Esempio
la funzione virtuale calcola non è definita in conversione, solo quando conversione è ereditata da una classe derivata è possibile creare un tipo completo

```
#include <iostream>
using namespace std;
class conversione
{
protected:

   double valore1; //valore iniziale
   double valore2; //valore convertito
public:
   conversione(double i)

   {

      valore1=i;

   }
   double prendi_convertito() {return valore2;}
   double prendi_iniziale() {return valore1;}
   virtual void calcola() =0; // virtuale pura
};
```

```
// litri in galloni

class litri_galloni: public conversione

{

public:

    litri_galloni(double i): conversione(i) {}

    void calcola()

    {

        valore2=valore1/3.7854;

    }

};

//      Fahrenheit        in        Celsius
class faranheit_celsius        :        public conversione
{

public:

    faranheit_celsius (double i) : conversione(i) {}

    void calcola()

    {

        valore2=(valore1-32)/1.8;

    }

};

int main()

{

    conversione *p; // puntatore alla classe base
    litri_galloni lgob(4);
    faranheit_celsius fcob(32);

    // per la conversione usa una funzione virtuale
    p=&lgob;
    cout<<p->prendi_iniziale()<< " litri sono ";

    p->calcola();

    cout <<p->prendi_convertito() <<" galloni \n";

    p=&fcob;

    cout<<p->prendi_iniziale()<< " in gradi fahreneit vale ";
```

```
    p->calcola();

    cout <<p->prendi_convertito() <<" in gradi celsius \n";

    return 0;

}

/*

 4 litri sono 1.05669 galloni

 32 in gradi fahreneit vale 0 in gradi celsius

 Press any key to continue

 */
```

ESERCIZIO 9
Classe astratta

```
#include <iostream>
using namespace std;
class persona {
private:
    int ID;
public:
    virtual void visualizzaID(int x)=0;
    friend int reID(class persona &);
    void print(){cout<<ID;}
};
class nonno:public persona{
private:
    char N[3];
public:
    static int cont; //non la conta nella sizeof
    friend void valore(nonno &);
    void contatore(){
        cout<<"\n\n contatore "<<cont<<"\n";
    }
    void virtual stampa(){
        printf("\n Dimensione di nonno = %ld\n",sizeof(class nonno));
                }
                void visualizzaID(int x){
                    printf("\n ID nonno %d \n",x);
                }
                //costruttori
    nonno(){cont++;}
                ~nonno(){cont--;}
                };

                class padre:virtual public nonno{ //manca il virtual per fare l'ereditierietà multipla
                private:
                    char P[3];
                public:
                    void stampa(){
```

```
        printf("\n Dimensione di padre= %ld \n",sizeof(class padre));
    }
    void visualizzaID(int x){
        printf("\n ID padre %d \n",x);
    }
    padre(){cont++;}
    ~padre(){cont--;}
};

class madre:virtual public nonno{ //manca il virtual per fare l'ereditierietà multipla
private:
    char M[3];
public:
    void stampa(){
        printf("\n Dimensione di madre= %ld \n",sizeof(class madre));
    }
    void visualizzaID(int x){
        printf("\n ID madre %d \n",x);
    }
    madre(){cont++;}
    ~madre(){cont--;}
};

class bamb:public madre,public padre{ //ricordare di mettere public quando
serve
private:
    char B[3];
public:
    void stampa(){
        printf("\n Dimensione di bambino= %ld \n",sizeof(class bamb));
    }
    void visualizzaID(int x){
        printf("\n ID bambino %d \n",x);
    }
    bamb(){cont++;}
    ~bamb(){cont--;}
};

int nonno::cont=0;

int main(){
    nonno *P;
    nonno NN;
    P=&NN;
    P->contatore();
    P->stampa(); // chiama la stampa del nonno
    P->visualizzaID(reID(NN));
    NN.print();

    padre PP;
    P=&PP;
    P->contatore();
    P->stampa(); // chiama quella ridefinita
    P->visualizzaID(reID(PP));
    PP.print();

    madre MM;
```

```
    P=&MM;
    P->contatore();
    P->stampa(); // chiama quella ridefinita
    //((madre*)P->stampa(); LA CONVERSIONE,CAST E' IMPLICITA
    P->visualizzaID(reID(MM));
    MM.print();

    bamb BB;
    P=&BB;
    P->contatore();
    P->stampa(); // chiama quella ridefinita
    P->visualizzaID(reID(BB));
    BB.print();
    system("PAUSE");
    return 0;

}
void valore(nonno &A){ A.N[0]='a';
}
int reID(persona &A){
    //la uso per passare i membri delle classi base nelle derivate
    A.ID=0;
    return A.ID;
}

//OUTPUT
// 12  24      24      36
```

PUNTATORI AD OGGETTI

Così come è possibile definire puntatori ad altri tipi di variabili,è possibile anche definire puntatori ad oggetti. Quando si deve accedere ad un membro di una classe con un puntatore ad oggetti si utilizza l'operatore **freccia "→"** in luogo dell'operatore **punto " . "**.
E' possibile utilizzare un puntatore padre per accedere alle variabili figlio.
p=&f; //per utilizzare le variabili membro devo fare il cast
((figlio*)p)->funzionefiglio();
(figlio*)p)->funzionepadre(); // oppure p=funzionepadre();
Il cast serve per far capire lo spiazzamento delle variabili del figlio rispetto a quelle del padre,che sono già presenti.
I puntatori ai membri non sono dei veri puntatori perchè forniscono il valore di scostamento all'interno della classe dove trovare la variabile(esempio 4).
L'utilizzo dei puntatori ad oggetti deve sempre e comunque rispettare l'**aritmetica dei puntatori(vedi esempio 2)**. Lo scostamento quindi si basa sul tipo dell'elemento puntato al momento della dichiarazione ed incrementare un puntatore vuol dire puntare all'elemento

successivo dello stesso tipo. Inoltre si può assegnare ad un puntatore l'indirizzo di un membro pubblico di un oggetto, in questo modo utilizziamo il puntatore per accedere a tale membro (vedi esempio 3).

```
Class persona{
        public:
int ID;
int assegna(int i){ID=i;}
};
int persona::*punt_dato; //puntatore a membro dato
int persona::*punt_funz)(); //puntatore a funzione membro.
punt_dato=&persona::ID; //calcola il valore di scostamento
punt_funz=&persona::assegna;
persona p;
cout<<p.*punt_dato; //si può fare perchè ID è public
cout<<(p.*punt_funz)(5);
persona *punt; //se uso i puntatori.
punt=&p;
cout<<punt->*punt_dato;
cout<<(punt->*punt_funz)(5);
```

Esempio 1:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) {i = j;}
int get_i() {return i;}
};
int main()
{
   cl ob(88), *p;
   p = &ob; // legge l'indirizzo di ob
   cout << p->get_i()<<endl; // usa -> per richiamare get_i()
   return 0;
}
```

Esempio 2

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl() {i = 0;}
cl(int j) {i = j;}
int get_i() {return i;}
};

int main()
{
   cl ob[3] = {1, 2, 3};
```

```
    cl *p;
    int i;
    p = ob; // punta all'inizio dell'array
    for (i=0; i<3; i++) {
    cout << p->get_i() << "\n";
    p++; // punta all'oggetto successivo
}
return 0;
}
```

Esempio 3

```
#include <iostream>
using namespace std;
class cl {
public:
int i;
cl(int j) {i = j;}
};
int main()
{
    cl ob(1);
    int *p;
    p = &ob.i; // legge l'inizio di ob.i
    cout << *p<<endl; // accede a ob.i tramite p
    return 0;
}
```

Esempio 4

```
#include <iostream>
using namespace std;
class cl {
public:
```

```
cl(int i) {val = i;};
int val;
int doppio_val() {return val+val;}
};
int main()
{
int cl::*data; // puntatore a membro dati
int (cl::*func)(); // puntatore a membro funzione
cl oggetto1(1),oggetto2(2); // crea oggetti
data=&cl::val; // calcola il valore di scostamento di val
func=&cl::doppio_val; // calcola il valore di scostamento di
doppio_val
cout<<"i valori:";
cout<<(oggetto1.*func)()<<" ";
cout<<(oggetto2.*func)()<<" "<<endl;
return 0;
}
```

ESERCIZIO 10
puntatori ad oggetti

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

class persona{
private:
   int ID;
public:
   void visualizza(){cout<<"ID: "<<ID<<"\n";}
   persona(){ID=0;}
   ~persona(){}
};

int main(){
   //istanzia l'oggetto di classe persona
   persona S; //alloca tutta la classe e chiama il costruttore
   persona *P; //istanzia la variabile puntatore a una classe di tipo persona ed alloca
   // i 4 byte per memorizzare il puntatorenello stack
   P=&S; // associo a P l'indirizzo di s ed inizializzo il puntatore
   cout<<"\nAccedendo alla variabile tramite l'operatore punto: ";

   S.visualizza();
   cout<<"\nAccedendo alla variabile tramite l'operatore freccia: ";
   P->visualizza();
   system("PAUSE");
   return 0;
}

/* ACCEDO ALA STESSA VARIABILE PRIVATA DELLA CLASSE utilizzando l'operatore
PUNTO(anticipato rispetto alla compilazione)


 ANTICIPATO
```

```
 COMPILE TIME,BINDING PRECOSE O

 utilizzando l'operatore FRECCIA (ritardato rispetto alla compilazione)
 RUNTIME,BINDING TARDIVO O RITARDATO
 */
```

Esempio di puntatori a tipi derivati

```
#include <iostream>
using namespace std;
class base {
int i;
public:
void set_i (int num) {i = num;}
int get_i() {return i;}
};
class derived: public base {
    int j;
public:
    void set_j(int num) {j=num;}
    int get_j() {return j;}
};
int main()
{
    base *bp;
    derived d;
    bp = &d; // il puntatore base punta all'oggetto derivato
    // accesso all'oggetto derivato utilizzando il puntatore base bp->set_i(10);
    cout << bp->get_i() << " " <<endl;
    /* questo non funziona. Non è possibile accedere a un elemento
     di una classe derivata
     utilizzando un puntatore alla classe base.

     bp ->set_j(88); // errore
     cout << bp->get_j(); // errore
     */
    return 0;
}
/* accesso consentito grazie al cast
 ((derived *)bp)->set_j(88);
 cout <<((derived *)bp)->get_j();*/
```

ESERCZIO 11
puntatori a classi PADRE e FIGLIO

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

class persona{
private:
    int ID;
public:
    void p_visualizza(){cout<<ID<<"\n";}
    persona(){ID=0;}
    ~persona(){}
};
```

```
class studente:public persona{
private:
   int matricola;
public:
   studente(){matricola=111111;}
   ~studente(){}
   void s_visualizza(){printf(" %d\n",matricola);}
};

int main(){
   persona *P; //dichiaro un puntatore a persona,classe padre


   persona Pe; //istanzio l'oggetto di tipo persona
   studente *S; //puntatore a studente,classe figlio
   studente St; //istanzio l'oggetto di tipo studente
   P=&Pe; //assegno l'indirizzo dell'oggetto padre alpuntatore d tipo padre
   cout<<"\nID persona ";
   P->p_visualizza(); // funziona,visualizza l' ID
   S=&St;
   cout<<"\nmatricola studente";

   S->s_visualizza();
   cout<<"\nID classe studente ";
   S->p_visualizza();
   printf("\n");
   //da errore di tipo cast        poichè il puntatore e l'indirizzo che associo sono diversi
   // E' una protezione del C++ ,x ovviarla faccio il casting
   //non funziona perchè usa lo spiazzamento del padre, infatti p è di tipo padre
   //Un puntatore di tipo base può essere usato per
   //puntare ad un oggetto derivato
   //Non vale la regola inversa

   P=&St; //al puntatore padre associo l'indirizzo del figlio
   P->p_visualizza(); //posso solo usare le funzioni del padre
   //P->s_visualizza(); //non è membro di persona...ok
   printf("\n");
   //mi sembra la stessa cosa...lo fa automaticamente il cast???
   P=&St; //assegno al puntatore tipo padre l'indirizzo di una classe figlio con cast in padre
   P->p_visualizza();//St diventando di tipo persona NON ammette le funzioni di studente
   //P->s_visualizza(); //non èun membro di persona...ok
   //P=&St; //o anche P=&((class persona)St);
   // ((studente*)P)=&St; //questo CAST non funziona
   // ((studente*)P)->p_visualizza(); //faccio cast P->variabile membro
   // ((studente*)P)->s_visuaizza(); // col cast si può accedere alle funzioni della classe
figlia

   //TUTTE LE ALTRE POSSIBILI COMBINAZIONI NON VALGONO
   //S=&Pe; errore di cast
   //S=&(studente)Pe;
   //(persona*)S=&Pe;
   // S=&((class s); ///Ovviamnte ERRORE
   system("PAUSE");
   return 0;
}
```

RIEPILOGO

1.puntatori a tipi derivati con costruttori standard

```
#include <iostream>

using namespace std;

class base {

private: int i; public:
   void set_i (int num) {i = num;}

   int get_i() {return i;}

};

class derived: public base {

private: int j; public:
   void set_j(int num) {j=num;}

   int get_j() {return j;}

};


 int main() // accesso ad una metodo di base tramite un base ed un base*

 {

 base b;

 base *bp;

 bp = &b; // il puntatore base* punta all'oggetto base b.set_i(11);
 bp->set_i(11);

 //cout <<b.i; errore i è privata

 //cout <<bp->i; errore i è privata cout <<b.get_i() << endl;
 cout << bp->get_i() << endl;

 return 0;

 }
```

```
        */
        /*
int main()
{
    base *bp; derived d; bp = &d;
    // il puntatore base punta all'oggetto derivato

    // accesso all'oggetto derivato utilizzando il puntatore base

    // il limite è per i metodi della derivata bp->set_i(22);
    cout << bp->get_i()<< endl;

    return 0;

        }*/
        /*
        int main() // accesso consentito grazie al cast di un
        puntatore base* a derived*
        // superamento limiti della derivata
        {
        base *bp;
        derived d;
        bp=&d;
        bp->set_i(22);
        cout << bp->get_i()<< endl;
        ((derived *)bp)->set_j(33);
        cout <<((derived *)bp)->get_j() <<endl;
        return 0;
        }
        */
        /*
```

```
int main() // accesso ad una metodo di base tramite un derived ed un derived*

{

    derived d;

    derived *dp = &d;

    d.set_i(44);

    cout <<d.get_i() <<endl;

    dp->set_i(55);

    cout <<dp->get_i() <<endl;

    //cout <<dp->i; errore i è privata di base return 0;
}

        */
        /*

int main() // se la derivazione è public non è necessario il cast fra derived* e base*
(diversamente dal contrario)

{

    derived d; derived *dp; dp=&d;
    base *b;

    b=dp;

    b->set_i(66);

    cout <<b->get_i() <<endl;

    return 0;

}
```

2.puntatori a tipi derivati con costruttori non standard

```
        /*
#include <iostream>

using namespace std;

class base {
```

```
private: int i; public:
   base (int x) {i=x; cout<< "costruzione di base" <<endl;}

   int get_i() {return i;}

};

class derived: public base {

private: int j; public:
   void set_j(int num) {j=num;}

   int get_j() {return j;}

};

int main() // accesso ad una metodo di base tramite un base ed un base*

{

   base b(11);

   base *bp;

   bp = &b; // il puntatore base* punta all'oggetto base

   //cout <<b.i; errore i è privata

   //cout <<bp->i; errore i è privata cout <<b.get_i() << endl;
   cout << bp->get_i() << endl;

   return 0;

}
        */
```

3. puntatori a tipi derivati con costruttori non standard

```
        /*
```

```
#include <iostream>
using namespace std;
class base {
private:

int i;

public:

base (int x) {i=x; cout<< "costruzione di base" <<endl;}

int get_i() {return i;}

};

class derived: public base {

private: int j; public:
   derived(int x):base(x) {cout <<"costruzione di derivata"
      <<endl;}

   void set_j(int num) {j=num;}

   int get_j() {return j;}

};
int main()

{

   base *bp; derived d(22); bp = &d;
   // il puntatore base punta all'oggetto derivato

   // accesso all'oggetto derivato utilizzando il puntatore base

   // il limite è per i metodi della derivata cout << bp->get_i()<< endl;
   return 0;

}
      */
      /*
```

```
#include <iostream>
using namespace std;
class base {
private:

int i;

public:

base (int x) {i=x; cout<< "costruzione di base" <<endl;}

int get_i() {return i;}

};

class derived: public base {

private: int j; public:
   derived(int x):base(x) {cout <<"costruzione di derivata"
      <<endl;}

   void set_j(int num) {j=num;}

   int get_j() {return j;}

};

int main() // accesso consentito grazie al cast di un puntatore base* a derived*

// superamento limiti della derivata

{

   base *bp; derived d(33); bp=&d;
   cout << bp->get_i()<< endl;

   ((derived *)bp)->set_j(44);

   cout <<((derived *)bp)->get_j() <<endl;

   return 0;

}
         */
```

```
        /*
#include <iostream>
using namespace std;
class base {
private: int i; public:
base (int x) {i=x; cout<< "costruzione di base" <<endl;}

int get_i() {return i;}

};
class derived: public base {

private: int j; public:
    derived(int x):base(x) {cout <<"costruzione di derivata"
        <<endl;}

    void set_j(int num) {j=num;}

    int get_j() {return j;}

};
int main() // accesso ad una metodo di base tramite un deriveded un derived*

{

    derived d(55);

    derived *dp = &d;

    cout <<d.get_i() <<endl;

    cout <<dp->get_i() <<endl;

    //cout <<dp->i; errore i è privata di base return 0;
    }
        */
#include <iostream>

using namespace std;

class base {

private: int i; public:
    base (int x) {i=x; cout<< "costruzione di base" <<endl;}

    int get_i() {return i;}

};

class derived: public base {
```

```
private: int j; public:
    derived(int x):base(x) {cout <<"costruzione di derivata"
        <<endl;}

    void set_j(int num) {j=num;}

    int get_j() {return j;}

};

int main() // la derivazione deve essere public

{

    derived d(77); derived *dp; dp=&d;
    base *b;

    b=dp;

    cout <<b->get_i() <<endl;

    return 0;

}
```

4. Puntatori a tipi derivati con costruttori non standard

Esempio1

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

class base {

private:
   int i;
public:
   base (int x) {

      i=x;
      cout << "costruzione di base" <<endl;

      cout << "indirizzo di i " << &i << endl;

      cout << "indirizzo di x " << &x << endl <<endl;

   }
   int get_i() {return i;}

   int k;

};

class derived: public base {

private:
   int j;
public:
   derived(int x):base(x) {cout <<"costruzione di derivata"
      <<endl;

      cout << "indirizzo di j " << &j << endl <<endl;}

   void set_j(int num) {j=num;}

   int get_j() {return j;}

};
int main()

{

   base b(11),b1(12);

   derived d(22);
```

```
    cout << "indirizzo oggetto base " << &b <<endl;

    cout << "indirizzo oggetto base " << &b1 <<endl <<endl; cout << "indirizzo oggetto
derivata " << &d <<endl; return 0;
}
```

Esempio 2

```
#include <iostream>
using namespace std;
class base {
private:

int i;
public:
    base() {}
base (int x) {

    i=x; cout << "costruzione di base" <<endl <<endl;

    cout << "indirizzo di i in base " << &i << endl;

    cout << "indirizzo di x in base " << &x << endl <<endl;

    cout << "valore di i in base " << i << endl;

    cout << "valore di x in base " << x << endl <<endl;

}

int j;

int get_i() {return i;}

int* get_ind_i() {return &i;}
    int* get_ind_j() {return &j;}
    int set_i() {return i=5;}
};

class derived: public base {

private:

    int k;

public:

    derived(int x) { k=x;

        cout <<"costruzione di derivata" <<endl <<endl;

        cout << "indirizzo di k di derived in derived " << &k <<
        endl <<endl;}

    void set_j(int num) {j=num;}

    int get_k() {return k;}
```

```
};

int main()

{

    base b(11);

    b.j=100;

    cout << "indirizzo oggetto di classe base " << &b <<endl
    <<endl;

    cout << "ora j in base vale "<< b.j << endl <<endl;

    derived d(22);

    cout << "indirizzo oggetto di classe derived " << &d
    <<endl <<endl;

    cout << "valore di inizializzazione per k in oggetto d " <<
    d.get_k() << endl <<endl;

    cout << "valore di inizializzazione per i in oggetto d " <<
    d.get_i() << endl <<endl;

    cout << "indirizzo di i derivato da base in oggetto d " <<
    d.get_ind_i() << endl <<endl;

    cout << "indirizzo di j derivato da base in oggetto d " <<
    d.get_ind_j() << endl <<endl;

    cout << "valore di i derivato da base in oggetto d " <<
    d.set_i() << endl <<endl;

    d.j=200;

    cout << "valore di j derivato da base in oggetto d " << d.j <<
    endl <<endl;

    return 0;

}
```

OUTPUT

```
costruzione di base
indirizzo di i in base 1245048
indirizzo di j in base 1245052
valore di i in base 11
valore di j in base -858993460
indirizzo oggetto di classe base 1245048
ora j in base vale 100
costruzione di derivata
indirizzo di k di derived in derived 1245044
indirizzo oggetto di classe derived 1245036
valore di inizializzazione per k in oggetto d 22
valore di inizializzazione per i in oggetto d -858993460
indirizzo di i derivato da base in oggetto d 1245036
indirizzo di j derivato da base in oggetto d 1245040
valore di i derivato da base in oggetto d 5
valore di j derivato da base in oggetto d 200
                  Press any key to continue
costruzione di base
indirizzo di i in base 1245048
indirizzo di j in base 1245052
valore di i in base 11
valore di j in base -858993460
indirizzo oggetto di classe base 1245048
ora j in base vale 100
costruzione di derivata
indirizzo di k di derived in derived 1245036
indirizzo oggetto di classe derived 1245032
valore di inizializzazione per k in oggetto d 22
```

```
valore di inizializzazione per i in oggetto d -858993460
indirizzo di i derivato da base in oggetto d 1245040
indirizzo di j derivato da base in oggetto d 1245044
valore di i derivato da base in oggetto d 5
valore di j derivato da base in oggetto d 200
                  Press any key to continue
```

PROBLEMA DEL COSTRUTTORE DI COPIA

I costruttori di copia sono particolari costruttori che vengono eseguiti quando un oggetto é creato per copia. Un costruttore di copia deve avere un solo argomento, dello stesso tipo dell'oggetto da costruire; l'argomento (che rappresenta l'oggetto esistente) deve essere dichiarato const (per sicurezza) e passato by reference (altrimenti si andrebbe a creare una copia della copia!).
point::point(const point& q) {.....}
e viene chiamato automaticamente ogni volta.
Per esempio, se definiamo un oggetto p e lo inizializziamo con un oggetto preesistente q:
point p = q ;
questa istruzione aziona il costruttore di copia, a cui é trasmesso q come argomento.
I costruttori di copia, come ogni altro costruttore, non sono obbligatori: se una classe non ne possiede, il C++ fornisce un costruttore di copia di default che esegue la copia membro a membro. Questo può essere soddisfacente nella maggioranza dei casi. Tuttavia, se la classe possiede dei membri puntatori, l'azione di default copia i puntatori, ma non le aree puntate: alla fine si ritrovano due oggetti i cui rispettivi membri puntatori puntano alla stessa area. Ciò potrebbe essere pericoloso, perché, se viene chiamato il distruttore di uno dei due oggetti, il membro puntatore dell'altro, che esiste ancora, punta a un'area che non esiste più.
Nell'esempio seguente una classe di nome A contiene, fra l'altro, un membro puntatore a int e un costruttore di copia che esegue le operazioni idonee ad evitare l'errore di cui sopra:

CLASSE	COSTRUTTORE DI COPIA

class A {
int* pa;
public:
A(const A&);
..... };
	A::A(const A& a){
	pa = new int ;
	*pa = *a.pa ;
	}

in questo modo, a seguito della creazione di un oggetto a2 per copia da un esistente oggetto a1:

A a2 = a1;

il costruttore di copia fa si che la variabile puntata *a1.pa venga copiata in *a2.pa; senza il costruttore sarebbe copiato il puntatore a1.pa in a2.pa.

Non esiste il distruttore di copia!

Un altro esempio sui costruttori di copia:

```
class persona{
private:
        int *p;
public:
 persona(int*){ p=new int [N];
for (i=0;i<N;i++)
p[i]=i;
}
~persona(){delete[] p;}
void funzione (persona P){
        printf("\nCIAO\n");
}
};

int main(){
        persona P(5);
        funzione(P);
        printf("\nCIAO\n");
return 0;
}

//OUTPUT     CIAO CIAO
                ERRORE!!!!!!!!!!!!
```

il compilatore dà errore solo alla chiusura del main perchè va a distruggere la seconda volta un'area di memoria che non esiste +.

ESERCIZIO 12
problematica del costruttore di copia e risoluzione

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

class vettore
{ private: int R;
   int *VETT;
public:
   //void assegna(class vettore) {cout<<"vediamo!!!!!";} //l'ho esclusa perchè non si
distingue
   void assegna(class vettore &) {cout<<"vediamo!!!!";}//per reference
   void assegna(class vettore *) {cout<<"vediamo!!!!";}//per indirizzo
   void dati()//è nella classe e non ha bisogno di dati parametrizzati.Se{
   {
      int i;
      for (i=0 ; i<R ; i++)
      {
         cout<<"\nInserire l'elemento VETT["<<i<<"]";
         cin>>VETT[i];
      }
   }
   vettore (int x) //costruttore
```

```
    {
        int i; R=x;
        cout<<"\ncostruttore del vettore\n"; VETT=new int[R];
        for(i=0;i<x;i++) VETT[i]=0;
    }
    ~vettore()
    {
        cout<<"\n\n distruttore vettore\n";
        delete[]VETT; //le parentesi quadre per far capire che è un vettore
    }
    void visualizza()
    {
        int i;
        for(i=0;i<R;i++)
            cout<<VETT[i];
    }
};

int main()
{ vettore v(3);
    v.dati();
    v.visualizza();
    //v.assegna(v); //questa cosa dà errore al return dal main
    v.assegna(v); //per reference funziona v.assegna(&v); //per indirizzo funziona
    return 0;
        }
```

```
/* OUTPUT DEL PROGRAMMA PASSANDO PER COPIA

”costruttore del vettore”
dati inseriti
visualizzazione dei dati
“distruttore del vettore”

“distruttore vettore”
“press any key to continue
ERRORE!!!! */
```

ESERCIZIO SU SIDE-EFFECT E REFERENCE

```
#include <iostream>
using namespace std;
class tempo
{
public:
   tempo(); // costruttore senza argomenti
   tempo(int);
   tempo( int, int, int ); // costruttore ore-minuti-secondi
   tempo( int, int ); // costruttore ore-minuti tempo( int ); // costruttore ore
   ~tempo(){cout<<" chiamata al distruttore " <<endl<<endl;}
   tempo UnOraPiuTardi();
   void UnOraPiuTardi_reference(tempo &);
   void AvanzaUnOra();
   void AvanzaUnOraErrataConst() const;
   void StampaSec() const;
   void PrintSec();
   static tempo OraDiPranzoStatic();
protected:
   // l'ora del giorno e' rappresentata come numero
   // dei secondi a partire dalla mezzanotte
   long int sec;
};
tempo::tempo()
{ sec = 0;
   cout<<"costruttore di default senza parametri"<<endl<<endl;
}
// DEFINIZIONE ALTERNATIVA PER EFFETTUARE TEST
//{ sec = 0; cout << "COSTRUTTORE SENZA ARGOMENTI\n"; }
tempo::tempo( int o, int m, int s )
{
   if( o < 0 || o > 24 || m < 0 || m > 60 || s < 0 || s > 60 )
```

```
        sec = 0;
    else sec = long(o)*3600 + m*60 + s;
    cout<<"costruttore con tre parametri"<<endl<<endl;
}
tempo::tempo( int o, int m )
{
    if( o < 0 || o > 24 || m < 0 || m > 60 )
        sec = 0;
    else
        sec = long(o)*3600 + m*60;
    cout<<"costruttore con due parametri"<<endl<<endl;
}
tempo::tempo( int o )
{
    if( o < 0 || o > 24)
        sec = 0;
    else
        sec = long(o)*3600;
    cout<<"costruttore con un parametro"<<endl<<endl;
}
// restituisce l'oggetto corrispondente ad un'ora piu'
// tardi rispetto all'oggetto di invocazione
tempo tempo::UnOraPiuTardi()
{
    tempo aux;
    aux.sec = ( sec + 3600) % 86400;
    return aux;
}
// esegue side-effect sull'oggetto di invocazione,
// aggiungendogli un'ora
void tempo::AvanzaUnOra()
{ sec = ( sec + 3600) % 86400; }
//void tempo::AvanzaUnOraErrataConst() const
//CC: "tempo.cpp", line 52: Error: The left operand cannot be assigned to.
//g++: tempo.cpp:52: assignment of read-only member `long int tempo::sec'
//{ sec = ( sec + 3600) % 86400; }
void tempo::PrintSec()
{ cout << "sono passati " << sec << " secondi" <<endl
    <<endl; }
void tempo::StampaSec() const
{ cout << "a mezzanotte " << " sono passati " << sec << " secondi" <<endl <<endl; }
tempo tempo::OraDiPranzoStatic()
{ return tempo(13,0); }




void tempo::UnOraPiuTardi_reference(tempo &tempo1)
{
    tempo1.sec=tempo1.sec+3600;
}
int main()
{
```

```
    tempo tempo0;
    tempo nonvogliodirticheoresono(25,76,-3);
    tempo tempo1(1,1,1); tempo tempo2 (1,1); tempo tempo3 (1); tempo tempo4;
    cout <<"fine creazione oggetti"<<endl << endl <<endl;
    tempo0.PrintSec();
    tempo1.PrintSec(); tempo2.PrintSec(); tempo3.PrintSec();
    nonvogliodirticheoresono.PrintSec();
    tempo1.AvanzaUnOra(); tempo1.PrintSec(); tempo2.UnOraPiuTardi();
tempo2.PrintSec(); tempo2.UnOraPiuTardi_reference(tempo2); tempo2.PrintSec();
    tempo4 = tempo0.UnOraPiuTardi(); tempo0.PrintSec(); tempo4.PrintSec();
    tempo1=tempo4;
    tempo1.PrintSec(); tempo4=tempo1.OraDiPranzoStatic(); tempo4.PrintSec();
    const tempo mezzanotte(24);
    mezzanotte.StampaSec();
    return 0;
}
```

OUTPUT

```
/* risultato
costruttore di default senza parametri
costruttore con tre parametri
costruttore con tre parametri
costruttore con due parametri
costruttore con un parametro
costruttore di default senza parametri
fine creazione oggetti
```

```
sono passati 0 secondi
sono passati 3661 secondi
sono passati 3660 secondi
sono passati 3600 secondi
sono passati 0 secondi
sono passati 7261 secondi
costruttore di default senza parametri
chiamata al distruttore
chiamata al distruttore
sono passati 3660 secondi
sono passati 7260 secondi
costruttore di default senza parametri
chiamata al distruttore
chiamata al distruttore
sono passati 0 secondi
sono passati 3600 secondi
sono passati 3600 secondi
costruttore con due parametri
chiamata al distruttore sono
passati 46800 secondi
costruttore con un parametro
a mezzanotte sono passati 86400 secondi
chiamata al distruttore
chiamata al distruttore
chiamata al distruttore
chiamata al distruttore
chiamata al distruttore
chiamata al distruttore
chiamata al distruttore
Press any key to continue
/*
```

COSTRUTTORE DI COPIE DI DEFAULT

Fa una copia bit a bit.
Definisco un costruttore di copie che faccia la copia e non lasci il puntatore "appeso".
Il costruttore di copie si basa sul concetto di reference.
esplicita->reference
implicita->puntatori
nome classe(const nome classe &nome){}

Es:

```
persona(const persona &ob){
        ID=new int;
        *ID=*ob.ID;
```

}
Crea un intero e associa a p,puntatore,l'indirizzo dell'area allocata.
Associa a quell'indirizzo,ovvero scrive nell'area puntata da P,quello che sta nell'area puntata da ID,con ID intero allocato nella classe (esempio1).
Per i vettori e matrici è la stessa cosa,ricordando che il nome è già l'indirizzo (C like).
Quindi creo una copia e non sovrascrivo i dati della classe.
Si fa l'overload del costruttore di copia quando un oggetto è inizializzato da un altro oggetto.
Si usa l'overload del costruttore di copia quando si usa la memoria dinamica.
1.Assegnamento
Persona p;
B=P; //non si usa il costruttore di copia.
2.Inizializzazione nella dichiarazione
persona B=P;
3.L'oggetto restituito da una funzione deve essere creato
Ha a che fare con gli oggetti temporanei.
4.Passaggio di un oggetto per copia dell'argomento di una funzione
Es:
Una funzione del tipo
persona funzione(persona B){
return B;
}
richiamerebbe 2 volte il costruttore di copia,una volta x l'argomento una volta per restituire.

Esempio 1

```
#include <iostream>
using namespace std;
class cl {
private:
int id; public: int i; cl(int );
// costruttore di copie
   cl (const cl &);
~cl();
int opposto(cl oggetto) {return -oggetto.id;}
};
cl::cl(int num)
{
   cout <<"costruzione di " <<num<< "\n";
   id=num;
}
cl::cl (const cl &oggetto)
{
   id=oggetto.id;
}
cl::~cl()
{
   cout <<"distruzione di " <<id<< "\n";
}
int main()
{
   cl oggetto(-1);
   cl oggetto1(-2);
   //cl oggetto2=oggetto;
```

```
   //cl oggetto3=oggetto1;
   cl oggetto2(oggetto);
   cl oggetto3(oggetto1);
   //oggetto.i=10;
   oggetto.i= oggetto.opposto(oggetto); cout<<"l'opposto vale " << oggetto.i<<"\n"; return
0;
}
```

OUTPUT

```
/* costruzione di -1
costruzione di -2
distruzione di -1
l'opposto vale 1
distruzione di -2
distruzione di -1
distruzione di -2
distruzione di -1
Press any key to continue/*
```

Funzione opposto realizzata con indirizzi ad oggetti

```
#include <iostream>
using namespace std;
class cl {
public:
   int i;
   cl(int i);
~cl();
void opposto(cl &oggetto) {oggetto.i=-oggetto.i;}
// non viene creato un oggetto temporaneo
};
cl::cl(int num)
{
   cout <<"costruzione di " <<num<< "\n";
   i=num;
}
cl::~cl()
{
   cout <<"distruzione di " <<i<< "\n";
}
int main()
{
   cl oggetto(-1);
   oggetto.i=10; oggetto.opposto(oggetto); cout<<oggetto.i<<"\n"; return 0;
}
```

OUTPUT

```
/*
costruzione di -1
-10
distruzione di -10
*/
```

Funzione opposto realizzata con puntatori a membri di una classe

```
#include <iostream>
```

```
using namespace std;
class cl {
public:
int i;
cl(int i);
~cl();
int opposto() { cout <<"l'opposto vale "; return -i;}
};
cl::cl(int num)
{
    cout <<"costruzione di " <<num<< "\n";
    i=num;
}
cl::~cl()
{
    cout <<"distruzione di " <<i<< "\n";
}
int main()
{
    int (cl::*funzione)(); cl oggetto(-1); funzione=&cl::opposto;
cout<<(oggetto.*funzione)()<<"\n"; return 0;
}
```

OUTPUT

```
/*
costruzione di -1
l'opposto vale 1
distruzione di -1
Press any key to continue
*/
```

Funzione opposto realizzata con puntatori ad oggetto

```
#include <iostream>
using namespace std;
class cl {
public:
    int i;
    int *p;
cl(int i);
~cl();
void opposto(int *p) { *p=-*p; }
};
cl::cl(int num)
{
    cout <<"costruzione di " <<num<< "\n";
    i=num;
};
cl::~cl()
{
    cout <<"distruzione di " <<i<< "\n";
}
int main()
{
    cl oggetto(-1);
    int *p;
```

```
    p=&oggetto.i; oggetto.opposto(p); cout<<*p<<endl; return 0;
}
```

OUTPUT

```
/*
costruzione di -1
1
distruzione di 1
Press any key to continue */
```

Funzione opposto realizzata con chiamata di funzione return

```
#include <iostream>
using namespace std;
class cl {
public:
int i;
cl(int i);
~cl();
int opposto(cl oggetto) {return -oggetto.i;}
};
cl::cl(int num)
{
    cout <<"costruzione di " <<num<< "\n";
    i=num;
}
cl::~cl()
{
    cout <<"distruzione di " <<i<< "\n";
}
int main()
{
    cl oggetto(-1);
    oggetto.i=10; oggetto.i=oggetto.opposto(oggetto); cout<<oggetto.i<<"\n";
    return 0;
}
```

OUTPUT

```
/*
```

```
costruzione di -1
distruzione di -1
1
distruzione di 1
Press any key to continue
*/
```

COSTRUTTORE PARAMETRIZZATO

Anche le funzioni costruttore possono essere parametrizzate come le funzioni generiche.

```
Class mia_classe{
        int a,b;
public:
mia_classe(int i, int j){
        a=i;
        b=j;
}
}
```

Questo codice consente di inizializzare a e b.

N.B. Il costruttore parametrizzato inibisce il costruttore di default.

Viene anch'esso definito inline.

Esempio 1

```
#include <iostream>
using namespace std;
class mia_classe { int a,b;
public:
mia_classe(int i,int j)
{
   a=i;
   b=j;
}
void mostra()
{
   cout <<a<<endl<<b<<endl;
}
};
int main()
{
   mia_classe prova(3,5);
   prova.mostra();
   return 0;
        }
```

Esempio 2

```
#include <iostream>
using namespace std;
class cl {
int h;
int i;
public:
```

```
cl (int j, int k) {h=j;i=k;} // costruttore con due parametri
   int get_i() {return i;}
int get_h() {return h;}
};
int main()
{
   cl ob[3] ={
      cl(1,2),cl(3,4),cl(5,6)
   }; // inizializzatori
   int i;
   for(i=0;i<3;i++)
   {
      cout<<ob[i].get_h();
      cout<<",";
      cout<<ob[i].get_i() <<"\n";
   }
   return 0;
}
```

METODI DI PASSAGGIO ALLE FUNZIONI

1.Per **copia.** Ridefinisco il costruttore di copia per non utilizzare quello di dafault.
2.Per **indirizzo.** Copio l'indirizzo in una variabile puntatore.
3.Per **reference.** Non viene creato il costruttore di copia.

Il passaggio per copia e quello per reference hanno la stessa chiamata e per questo motivo non si può fare l'overload di queste funzioni per il metodo ma si può fare solo per il numero di argomenti e tipo degli stessi.

VARIABILE STATICA

E' una variabile globale che vale solo nella classe in cui è dichiarata. Per dichiarare una variabile statica si utilizza la parola chiave **static**. Le variabili globali sono automaticamente inizializzate a 0;tutti gli oggetti della classe hanno accesso alle variabili static.
Static int a;
int persona::a=0 //fuori dal main

Esempio

```
#include <iostream>
using namespace std;
class condivisione {
   static int a;
int b;
public:
void imposta(int i,int j)
{ a=i; b=j;
}
void mostra();
};
int condivisione::a; // definisce a ovvero alloca memoria: è come una definizione globale
per la classe
void condivisione::mostra()
{
   cout <<"variabile statica a = "<<a<<endl;
   cout <<"variabile non statica b = "<<b<<endl;
```

```
}
int main()
{
   condivisione x,y;
   x.imposta(1,1); // assegna 1 alla variabile a x.mostra();
   y.imposta(2,2); // assegna 2 alla variabile a y.mostra();
   x.mostra();
   return 0;
}
```

OUTPUT

```
/* risultato
variabile statica a:1
variabile non statica b:1
variabile statica a:2
variabile non statica b:2
variabile statica a:2
variabile non statica b:1
*/
```

esempio di dati static per registrare il numero di oggetti all'interno di una classe

```
#include <iostream>
using namespace std;
class contatore {
public:
static int conta;
contatore()
{
   conta++;
}
~contatore()
{
   conta--;
}
};
int contatore::conta;
void f();
int main()
{
   contatore oggetto1;
   cout <<"oggetti esistenti:"; cout <<contatore::conta<<"\n"; contatore oggetto2;
   cout <<"oggetti esistenti:";
   cout <<contatore::conta<<"\n";
   f();
   cout <<"oggetti esistenti:";
   cout <<contatore::conta<<"\n";
   return 0;
}
void f()
{
   contatore temp; //temp viene distrutta all'uscita da f()
   cout <<"oggetti esistenti:";
   cout <<contatore::conta<<"\n";
}
```

OUTPUT

```
/*
Oggetti esistenti: 1
Oggetti esistenti: 2
Oggetti esistenti: 3
Oggetti esistenti: 2
*/
```

PASSAGGIO DI UN OGGETTO A UNA FUNZIONE E RESTITUZIONE DI UN OGGETTO DA PARTE DI UNA FUNZIONE

Esempio di passaggio di un oggetto a funzione

```
#include <iostream>
using namespace std;
class myclass
{
   int i;
public:
   myclass(int n);
   ~myclass();
   void set_i (int n) {i = n;}
   int get_i() {return i;}
};
myclass::myclass(int n)
{
   i = n;
   cout << "Costruzione di " << i << "\n";;
}
myclass:: ~myclass()
{
   cout << "Distruzione di " << i << "\n";
}
void f(myclass ob);
int main()
{
   myclass o(1);
   f(o);
   cout << "Questa la i di main: ";
   cout << o.get_i() << "\n";
   return 0;
}
void f(myclass ob)
{
   ob.set_i(2);
   cout << "Questa la i locale: " << ob.get_i() <<endl;
}
```

OUTPUT

```
/*
Costruzione di 1
Questa la i locale:2
Distruzione di 2
Questa la i di main: 1
Distruzione di 1
*/
```

Restituzione di un oggetto da una funzione

```
#include <iostream>
using namespace std;
class myclass
{
   int i;
public:
   void set_i (int n) {i = n;}
   int get_i() {return i;}
};
myclass f(); // restituisce un oggetto di tipo myclass
int main()
{
   myclass o;
   o = f();
   cout << o.get_i() << "\n";
   return 0;
   }
   myclass f()
   {
      myclass x; x.set_i(1); return x;
   }
```

Assegnamento di oggetti

```
#include <iostream>
using namespace std;
class myclass
{
   int i;
public:
   void set_i (int n) {i = n;}
   int get_i() {return i;}
};
int main()
{
   myclass ob1, ob2;
   ob1.set_i(99);
   ob2 = ob1;
   // assegna i dati da ob1 a ob2
   cout << "questa la i di ob2: " << ob2.get_i()<<endl;
   return 0;
}
```

Passaggio di un oggetto per indirizzo

```
#include <iostream>
using namespace std;
class cl {
int id; public: int i;
cl(int i);
~cl();
int opposto(cl oggetto) {cout <<"l'opposto vale"; return
   -oggetto.i;}
};
cl::cl(int num)
{
   cout <<"costruzione di " <<num<< "\n";
```

```
    id=num;
}
cl::~cl()
{
    cout <<"distruzione di " <<id<< "\n";
}
int main()
{
    cl oggetto(-1);
    oggetto.i=10; oggetto.i=oggetto.opposto(oggetto); cout<<oggetto.i<<"\n";
    return 0;
}
```

OUTPUT

```
/*
costruzione di -1
-10
distruzione di -1
*/
```

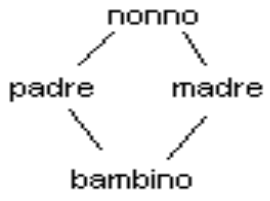
VETTORI AD OGGETTI

Se consideriamo la solita classe persona,
persona p[3]; è un vettore in cui ogni elemento è un oggetto.
Nella classe devo mettere sempre un costruttore senza parametri perchè se ne ho uno parametrizzato questo maschererà quello di default. Potrei anche utilizzare un costruttore che non fa nulla. Persona(){};
persona p[3]={persona(),persona(),persona()};
Devo inizializzare chiamando i costruttori parametrizzati. Se il costruttore ha un solo valore posso mettere direttamente il valore.

Gli oggetti allocati dinamicamente non possono essere inizializzati quindi devo avere il costruttore senza parametri per evitare errori di compilazione.

EREDITARIETA' MULTIPLA

Le classi devono essere almeno 4:



Nonno ha una sola variabile N;
Padre ha 2 variabili: N ereditata ; P specifica
Madre ha 2 variabili: N , M
Bambino ha N,P,N,M,B
Il bambino eredita sia dal padre che dalla madre e per questo ha 2 variabili N e quindi non posso accedervi perchè non saprei distinguerle.

Per risolvere questo problema ricorro all'**ereditarietà multipla**.
Dichiaro la classe madre e padre di tipo virtual. In questo modo la variabile N è trattata come una sola.
Quando due o più oggetti sono derivati da una classe base comune, è possibile evitare che in un oggetto derivato da questi oggetti, siano presenti più copie della classe base dichiarando la **classe base** come **virtual,** nel momento in cui viene ereditata, di seguito l'esempio:

```
#include <iostream>
using namespace std;
class base
{

public:

   int i;

};

class derivata1 : virtual public base

{ public: int j;
};

class derivata2 : virtual public base

{
```

```
public:
```

```
    int k;

};

class derivata3 :public derivata1, public derivata2

{

public:

    int somma;

};

int main()

{

    derivata3 ob; ob.i=10; ob.j=20; ob.k=30;
    ob.somma=ob.i+ob.j+ob.k; cout <<ob.somma<<endl; return 0;
}

/*

 60

 Press any key to continue

 */
```

Esempio con utilizzo di più classi base e costruttori

```
#include <iostream>
using namespace std;
class base1
{
protected: int i; public:
    base1(int x) {i=x;cout<<"costruzione di base1\n";}
    ~base1() {cout<<"distruzione di base1\n";}
};
class base2
{ protected: int k; public:
    base2(int x) {k=x; cout<<"costruzione di base2\n";}
    ~base2() {cout<<"distruzione di base2\n";}
};
// il costruttore della classe base deve dichiarare tuti i propri parametri e tutti quelli richiesti
// dalla classe base
class derivata :public base1, public base2
{
    int j;
public:
    derivata(int x, int y, int z):base1(y),base2(z)
    {j=x; cout<<"costruzione di derived\n";}
    ~derivata(){cout<<"distruzione di derivata\n";}
    void mostra() {cout <<i<<""<<j<<""<<k<<"\n"; }
};
int main()
```

```
{
   derivata ob(3,4,5);
   ob.mostra();
   return 0;
}
/*
 costruzione di base1 costruzione di base2 costruzione di derived
 435
 distruzione di derivata distruzione di base2 distruzione di base1
 Press any key to continue
 */
```

ESERCIZIO 13
ereditarietà multipla

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

class nonno{
private:
   char N[3];
public:
   static int cont; //non la conta nella sizeof
   friend void valore(nonno&);
   void contatore(){
      cout<<"\n\n contatore"<<cont<<"\n";
   }
   void virtual stampa(){
      printf("\n\n Dimensione di nonno= %d",sizeof(class nonno));
   }
   //costruttori
   nonno(){cont++;}
   ~nonno(){cont--;}
};

class padre:virtual public nonno{ //manca il virtual per fare l'ereditarietà multipla
private:
   char P[3];
public:
   void stampa(){
      printf("\n\n Dimensione di padre =%d",sizeof(class padre));
            }
            padre(){cont++;}
            ~padre(){cont--;}
            };

            class madre:virtual public nonno{ //manca il virtual per fare l'ereditarietà multipla
            private:
               char M[3];
            public:
               void stampa(){
                  printf("\n\n Dimensione di madre =%d",sizeof(class madre));
                        }
                        madre(){cont++;}
                        ~madre(){cont--;}
                        };
```

```
class bamb:public madre,public padre{ //ricordate di mettere PUBLIC
quando serve
private:
   char B[3];
public:
   void stampa(){
      printf("\n\n Dimensione di bambino = %ld",sizeof(class bamb));
          }
          bamb(){cont++;}
          ~bamb(){cont--;}
          };

          int nonno::cont=0;

          int main(){ nonno *P; nonno NN; P=&NN;
             P->contatore();
             P->stampa();// chiama la stampa del nonno
             padre PP; P=&PP;
             P->stampa(); //chiama quella ridefinita
             madre MM;
             P=&MM;
             P->stampa();// chiama quella ridefinita
             //((madre*)P)->stampa(); //conversione.Cast implicito
             bamb BB;
             P=&BB;
             P->contatore();
             P->stampa(); // problema compilatore?
             /*printf("nonno %d",sizeof(nonno));
              printf("madre %d",sizeof(madre)); printf("padre
%d",sizeof(padre)); printf("bamb %d",sizeof(bamb));*/
             /*NN.stampa();
              PP.stampa(); MM.stampa();
              BB.stampa();//chiamate senza puntatore*/
             // valore(BB);
             system("PAUSE");
             return 0;
          }
          void valore(nonno &A){
             A.N[0]='a';
          }

     /*metto una variabile static ed il costruttore per contare
incremento nel costruttore e decremento nel distruttore*/
       //sequenza nonno padre madre bamb senza considerare
       costruttori e distruttori
       //output 8 12 12 28 //per l'allineamento va di 4 in 4
       //senza la virtual 3 6 6 15
       //senza virtual con ereditarietà multipla 3 11 11 23
       //con virtual ed ereditarietà multipla 8 16 16 28
       //costruttori e distruttori influiscono solo sulle classi
       derivate x quanto riguarda le dimensioni
```

PROGETTO MODULARE

Nella creazione di un progetto è necessario creare almeno 3 file.
Un main.cpp con la funzione main.
Un file.cpp con la definizione delle funzioni membro delle classi.

Un file.h (header file) con la definizione della classe. Ovviamente uno per ogni classe.
I FILE header .h contengono le direttive di isolamento:
#ifndef NOMEMACRO //if not define
#define NOMEMACRO //define
“ “
classe
“
“
#endif

Servono per isolare le linee di codice. Le considero solo una volta nel progetto.

I FILE classe .cpp
Servono ad includere le librerie.

#include “nome.h” //il file dell'header a cui fa riferimento
Vi si mettono le funzioni,come se fossero offline,con il loro corpo.

Il FILE main.cpp
Oltre alle librerie include tutti i file .cpp e vi si scrive il main.
Per la composizione si scrive prima la classe contenitore.
Si mette prima l'header della classe contenitore poi quella della classe contenuta.

ESERCIZIO 14

```
//costruttori di copie per allocazione dinamica

//main.cpp
```

```
#include "ve.h"
#include "in.h"
#include "ma.h"

#include <iostream>
using namespace std;

int main(){
	Inte intero;//istanzio intero
	intero.assegna(20);
	visual_I(intero);


cout<<"-------------------------------------------------------
-";
	Vett VV(5);
	VV.assegna();
	visual_V(VV);


cout<<"-------------------------------------------------------
-";

	Matr MM(2,3);
	MM.assegna();
	visual_M(MM);
	system("PAUSE");
	return 0;
}

//in.h
#ifndef INTE_H
#define INTE_H
class Inte{
private:
	int *I;
public:
	void assegna (int A);
	Inte (const Inte & ob1);
	Inte();
	~Inte();
	friend void visual_I(Inte);
};
#endif
```

```cpp
//in.cpp
#include "in.h"

#include <iostream>
using namespace std;

Inte::Inte(){
	I=new int;
	cout<<"\n\n costruttore intero";
}
Inte::Inte(const Inte & ob1){ //costruttore di copia
	I=new int;
	*I=*ob1.I; //fa la copia
	cout<<"\n\n costruttore di copia intero";
}
void Inte::assegna(int A){
	*I=A; //serve l'operatore * per accedere all'area di
memoria di I
}
Inte::~Inte(){
	delete I;
	cout<<"distruttore intero";
}
void visual_I(Inte A){
	cout<<"\n\nL'intero\n\n"<<*A.I<<endl;
}

//ve.h
#ifndef VETT_H
#define VETT_H

class Vett{
private:
	int *V;
	int N;
public:
	void assegna();
	Vett(const Vett & ob2);
	Vett(int);
	~Vett();
	friend void visual_V(Vett);
};
#endif
```

```cpp
//ve.cpp
#include "ve.h"

#include <iostream>
using namespace std;

void Vett::assegna()
{
	for(int i=0;i<N;i++)
		V[i]=i;
}
Vett::Vett(const Vett & ob2){
	V=new int[ob2.N];
	for(int i=0;i<ob2.N;i++)
		V[i]=ob2.V[i]; //fa la copia
	N=ob2.N; //devo inizializzare tutte le variabili
	cout<<"\n\n costruttore di copia vettore\n\n";
}

Vett::Vett(int A=2){
	N=A;
	V=new int [N];
	cout<<"\n\n costruttore vettore";
	cout<<"\nN= "<<N<<"."<<endl;
}

Vett::~Vett(){
	delete []V;//distruttore vettore
	cout<<"\n\n distruttore vettore";
}

void visual_V(Vett A){
	cout<<"\n\n Vettore \n\n";
	for (int i=0;i<A.N;i++)
		cout<<"\n\n V["<<i<<"]: "<<A.V[i]<<endl;
}

//ma.h
#ifndef MATR_H
#define MATR_H

class Matr{
private:
	int **M;
	int R,C;
```

```
public:
	void assegna();
	Matr(const Matr & matrix);
	Matr(int,int);
	~Matr();
	friend void visual_M(Matr);
};
#endif

//ma.cpp
#include "ma.h"

#include <iostream>
using namespace std;

void Matr::assegna(){
	int cont=0;
	for(int i=0;i<R;i++)
		for(int j=0;j<C;j++)
		{
			M[i][j]=cont;
			cont++;
		}
}

Matr::Matr (const Matr & matrix){
	R=matrix.R;
	C=matrix.C;
	M= new int* [R];
	for (int i=0;i<R;i++)
		M[i]=new int [C];
	for (int i=0;i<R;i++) //la i l'ho dichiarata nel for di
prima
		for (int j=0;j<C;j++)
			M[i][j]=matrix.M[i][j];
	cout<<"\n costruttore di copie matrice \n ";
}

Matr::Matr(int A=2, int B=2){
	R=A;
	C=B;
	M=new int* [R];
	for (int i=0;i<R;i++)
		M[i]=new int [C];
	cout<<"\n costruttore matrice \n\n";
}
```

```
Matr::~Matr(){
	for (int i=0;i<R;i++)
		delete []M[i];
	delete []M;
	cout<<"\n\n distruttore matrice";
}

void visual_M(Matr A){
	cout<<"\n\n Matrice \n\n";
	for(int i=0;i<A.R;i++)
	{
		for(int j=0;j<A.C;j++)
			cout<<"M["<<i<<"]["<<j<<"]="<<A.M[i]
[j]<<endl;
	}
}
```

OVERLOAD DEGLI OPERATORI

In C++ è possibile eseguire l'overloading della maggior parte degli operatori. L'overloading degli operatori viene eseguito creando funzioni **operator**,che definiscono le specifiche operazioni che dovranno essere svolte dall'operatore modificato tramite overloading rispetto alla classe specificata. Le funzioni operator possono essere o meno funzioni membro della classe su cui operano. Quelle non membro sono quasi sempre funzioni friend della classe.

Creazione di una funzione operator membro:

TIPO-RESTITUITO NOME-CLASSE::operator#(elenco argomento)
{
//operazioni
}

Esempio di overloading di []

```
#include <iostream>
using namespace std;
class atype {
int a[3];

public:

atype(int i, int j, int k)
  {
  a[0] = i; a[1] = j; a[2] = k;
}

int operator[](int i) {return a[i];}

};

int main()

{
```

```
    atype ob(1,2,3);

    cout << ob[1] << endl; // visualizza 2 return 0;
}
```

Overloading di [] --- [] si puo' ora usare sia a sinistra che a destra di un assegnamento

```
#include <iostream>
using namespace std;
class atype {
int a[3];
public:

atype(int i, int j, int k) {

    a[0] = i; a[1] = j; a[2] = k;
}

int &operator[](int i) {return a[i];}

};

int main()

{

    atype ob(1,2,3);

    cout << ob[1]; // visualizza 2
    cout << "\n";

    // ora operator[]() restituisce un indirizzo all'elemento del vettore indicato da i

    // in questo caso la coppia [] può essere utilizzata anche sul lato sinistro di una
istruzione

    // di assegnamento in modo da modificare un elemento di un vettore

    ob[1] = 25; // [] alla sinistra di = cout << ob[1]; // ora visualizza 25 return 0;
}
```

un esempio di array sicuro

```
    // aggiunge la verifica dei limiti a ovl_op_part_2
#include <iostream>
#include <cstdlib>
using namespace std;
class atype {
int a[3];

public:

atype(int i, int j, int k) {

    a[0] = i; a[1] = j; a[2] = k;
```

```
}

int &operator[](int i);

};

// verifica i limiti per atype
int &atype::operator [](int i)
{
   if(i<0 || i>2) {

       cout << "Superamento dei limiti\n";

       exit(1);

       }

       return a[i];

       }

       int main()

       {

          atype ob(1,2,3);

          cout << ob[1]; // visualizza 2 cout << "\n";
          ob[1] = 25; // [] alla sinistra di =

          cout << ob[1] << endl; // visualizza 25

          ob[3] = 44; // genera un errore run-time perche' 3 e' oltre i limiti

          return 0;

       }
```

overload di ()

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) { longitude = lg; latitude = lt;
}
void show() {
   cout << longitude << " ";
   cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator()(int i, int j);
};

// Overloading di () per loc
```

```
loc loc::operator()(int i, int j)
{
   longitude = i;
   latitude = j;
   return *this;
}
// Overloading di + per loc
loc loc::operator+(loc op2)
{
   loc temp;
   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitude;
   return temp;
}
int main()
{
   loc ob1(10, 20), ob2(1,1);
   ob1.show();
   ob1(7,8); // puo' essere utilizzata da sola ...
   ob1.show();
   ob1 = ob2 + ob1(10,10); // ... o all'interno di espressioni
   ob1.show();
   return 0;
}
```

```
/*
10 20
7 8
11 11
Press any key to continue
*/
```

Altri esempi di overloading

```
#include <iostream>
using namespace std;
class myclass {
public:
int i;
myclass *operator ->() {return this;}
};
```

```
int main()
{
myclass ob;
ob ->i = 10; // uguale a ob.i
cout << ob.i << " " << ob->i;
return 0;
}
// overloading dell'operatore virgola
```

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;

public:

loc() {}

loc(int lg, int lt) { longitude = lg; latitude = lt;
}

void show() {

   cout << longitude << " ";

   cout << latitude << "\n";

}

loc operator+(loc op2);

loc operator,(loc op2);

};

// Overloading dell'operatore virgola per loc
loc loc::operator,(loc op2)
{

   loc temp;

   temp.longitude = op2.longitude;
```

```
    temp.latitude = op2.latitude;

    cout << op2.longitude << " " << op2.latitude << "\n";

    return temp;

    }

    // Overloading dell'operatore + per loc
loc loc::operator+(loc op2)

    {

        loc temp;

        temp.longitude = op2.longitude + longitude; temp.latitude = op2.latitude + latitude;
return temp;
    }

    int main()

    {

        loc ob1(10, 20), ob2(5,30), ob3(1,1);

        ob1.show(); ob2.show(); ob3.show(); cout << "\n";
        ob1 = (ob1, ob2+ob2, ob3);

        ob1.show(); // visualizza 1 1, il valore di ob3 return 0;
    }
        /*
        10 20
        5 30
        1 1
        10 60
        1 1
        1 1
```

```
Press any key to continue
*/
```

ESERCIZIO15
creazione di una funzione operator membro

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

class loc{
    int longitudine,latitudine;
public:
    loc(){}
    loc(int lg, int lt){
        longitudine=lg;
        latitudine=lt;
    }

    void visualizza(){
        cout<<"longitudine "<<longitudine<<" ";
        cout<<"latitudine "<<latitudine<<"\n";
    }

    loc operator+(loc op2);
};

//overloading di + per loc
loc loc::operator +(loc op2){
    loc temp; temp.longitudine=op2.longitudine+longitudine;
temp.latitudine=op2.latitudine+latitudine; return temp;
}

int main(){
    loc ob1(10,20),ob2(5,30);
    ob1.visualizza(); //visualizza 10 e 20 ob2.visualizza(); //visualizza 5 e 30 ob1=ob1+ob2;
    ob1.visualizza(); //visualizza 15 e 50
    system("PAUSE");
    return 0;
}
```

Creazione di operatori tramite funzioni friend:

ad una funzione di overloading **friend operator** gli operandi devono essere passati esplicitamente.
Operatore binario->2 parametri ,facendo attenzione a passare l'operando di sinistra come primo e quello di destra come secondo.
Operatore unario->1 parametro.

ESERCIZIO16
overloading di operatori tramite funzioni friend

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

class loc{
   int longitudine,latitudine;
public:
   loc(){} //necessario per costruire valori temporanei
   loc(int lg, int lt){ longitudine=lg; latitudine=lt;
   }

   void visualizza(){
      cout<<"longitudine "<<longitudine<<" ";
      cout<<"latitudine "<<latitudine<<"\n";
   }

   friend loc operator+(loc op1, loc op2); //ora è una friend
};

loc operator+(loc op1,loc op2){ loc temp;
temp.longitudine=op1.longitudine+op2.longitudine;
temp.latitudine=op1.latitudine+op2.latitudine; return temp;
}

int main(){
   loc ob1(10,20),ob2(5,30); ob1.visualizza(); //visualizza 10 e 20 ob2.visualizza();
//visualizza 5 e 30 ob1=ob1+ob2;
   ob1.visualizza(); //visualizza 15 e 50
   system("PAUSE");
   return 0;
}
```

E' possibile realizzare **l'overloading dei costruttori**:
consentono sia di garantire **maggiore flessibilità**, sia di creare **oggetti inizializzati e non inizializzati**, sia di creare costruttori di copie quando si crea una classe per la quale esistono due o più modi per costruire un oggetto è opportuno fornire una funzione costruttore modificata tramite overloading per questi diversi modi. Se si cerca di creare un oggetto per il quale non esiste un costruttore viene prodotto un errore in fase di compilazione

Primo esempio per garantire maggiore flessibilità

```
#include <iostream>
```

```
using namespace std;
class data {
   int giorno, mese, anno;
public:
   data (char *);
   data (int , int , int );
   void mostra_data();
};
// inizializzazione con una stringa
data::data(char *d)
{
   scanf(d, "%d%*c%d%*c%d", &giorno, &mese, &anno);
}
// inizializzazione con interi
data::data(int g, int m, int a)
{
   giorno=g; mese=m; anno=a;
}
void data::mostra_data()
{
   cout << giorno <<"/"<<mese<<"/"<<anno<<endl;
}
int main()
{
   char s[80];
   data oggetto1(20,02,1969), oggetto2("20/02/1969");
   oggetto1.mostra_data();
   oggetto2.mostra_data();
   cout <<"immettere nuova data"<<endl;
   cin >>s;

   data d(s); d.mostra_data(); return 0;
}
```

Secondo esempio per creare oggetti inizializzati e non inizializzati
N.B. non è possibile inizializzare array allocati dinamicamente
il seguente programma dichiara due array di tipo potenze: uno inizializzato e l'altro no

```
#include <iostream>
#include <new>
using namespace std;
// Il costruttore standard viene impiegato per costruire l'ar-ray non inizializzato di3 e
// l'array allocato dinamicamente;
// il costruttore parametrizzato viene richiamato per creare gli oggetti dell'array di2
class potenze {
   int x;
public:
   // overloading del costruttore in due modi
   potenze() {x=0;} // senza inizializzazione
   potenze(int n) {x=n;} // con inizializzazione
   int prendi_x() {return x;}
   void imposta_x(int i) {x=i;}
};
int main()
{
   potenze di2[]= {1,2,4,8,16}; // inizializzato
   potenze di3[5] ;// non inizializzato
```

```
    potenze *p;
    int i;
    // Visualizza le potenze di 2
    cout<<"potenze di 2:";
    for(i=0;i<5;i++)
    {cout<< di2[i].prendi_x()<<" ";}
    cout <<"\n\n";
    // imposta le potenze di 3 di3[0].imposta_x(1); di3[1].imposta_x(3); di3[2].imposta_x(9);
di3[3].imposta_x(27); di3[4].imposta_x(81);
    // visualizza le potenze di 3 cout<<"potenze di 3:";
    for(i=0;i<5;i++)
    {cout<< di3[i].prendi_x()<<" ";}
    cout <<"\n\n";
    // allocazione dinamica di un array
    //try
    //{ p=new potenze[5]; // nessuna inizializzazione
    //}
    //catch( bad_alloc xa)
    //{
    //cout <<"errore di allocazione";
    //return 1;
    //}
    // inizializza l'array dinamico con le potenze di due for(i=0;i<5;i++)
    {p[i].imposta_x(di2[i].prendi_x());}
    // Visualizza le potenze di 2
    cout<<"potenze di 2:";
    for(i=0;i<5;i++)
    {cout<< p[i].prendi_x()<<" ";}
    cout <<"\n\n";
    delete [] p;
    return 0;
}
```

Terzo esempio per creare costruttori di copie

N.B. quando si esegue una copia i due oggetti sono identici bit a bit, quando però la inizializzazione comporta l'allocazione di memoria non deve essere fatta una copia bit a bit

Questo programma crea una classe per array "sicuri"

```
#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;
class vettore
{
    int *p;
    int dimensione;
public:
    vettore (int dim)
    { cout<<"costruzione"<<endl; p=new int[dim]; dimensione=dim;
    }
    ~vettore() {delete [] p;cout<<"distruzione"<<endl;}
    // costruttore di copie
    vettore(const vettore &vett);
    void imposta (int i, int j)
    {
        if(i>=0 && i <dimensione)
            p[i]=j;
```

```
        }
    int prendi(int i)
    {
        return p[i];
    }
};
// Costruttore di copie
vettore::vettore(const vettore &vett)
{
    int i;
    p=new int[vett.dimensione]; cout<<"costruttore di copie"<<endl; for
(i=0;i<vett.dimensione;i++) p[i]= vett.p[i];
    }
    int main()
    {
        vettore num(10);
        int i;
        for(i=0;i<10;i++) num.imposta(i,i);
        for(i=9;i>=0;i--) cout << num.prendi(i);
        cout <<"\n";
        // i vettori num e x contengono gli stessi valori ma ciascun vettore
        // si trova in aree diverse di memoria
        // crea un altro vettore e lo inizializza con num
        vettore x(num); // richiama il costruttore di copie;
        for(i=0;i<10;i++) cout << num.prendi(i);
        cout <<"\n";
        return 0;
            }OUTPUT

        /*
        costruzione
        9876543210
        costruttore di copie
        0123456789
        distruzione
```

```
distruzione
Press any key to continue
*/
```

Quarto esempio per creare costruttori di copie: ricerca dell'indirizzo di una funzione modificata tramite overloading

```
#include <iostream>
using namespace std;
int mia_funzione(int);
int mia_funzione(int , int);
int main()
{
   int (*fp)(int a); // puntatore a int f(int)
   // nell'altro caso
   // int (*fp)(int a, int b);
   fp= mia_funzione; // punta a mia_funzione(int)
   cout <<fp(5)<<endl;
   return 0;
}
int mia_funzione(int a)
{
   return a;
}
int mia_funzione(int a, int b)
{
   return a*b;
}
```

```
/*
5
Press any key to continue
*/
```

OPERATORI UNARI

Nel C++ è possibile creare versioni prefisse e postfisse degli operatori incremento e decremento.

```
tipo operator ++() {//++ usato come prefisso
// corpo dell'operatore
}
tipo operator ++(int x){//++ è usato come postfisso
//corpo dell'operatore
}
```

ESERCIZIO 17
overload operatori

```
//tridim.h
#ifndef TRIDIM_H
#define TRIDIM_H

class tridim
{
private:
      int X;
      int Y;
      int Z;
public:
      static int cont;
      tridim(int,int,int);
      tridim();
      tridim operator+(tridim);
      tridim operator=(tridim);
      void visualizza();
      tridim(const tridim&);
      ~tridim();
};

#endif
```

```
//tridim.cpp
#include "tridim.h"

#include <iostream>
using namespace std;

tridim tridim::operator +(tridim ob)
{
      tridim temp;
      temp.X=X+ob.X;
      temp.Y=Y+ob.Y;
      temp.Z=Z+ob.Z;
      return temp;
}

tridim tridim::operator =(tridim ob)
{
```

```
	X=ob.X;
	Y=ob.Y;
	Z=ob.Z;
	return *this;
}

tridim::tridim(const tridim & ob)
{
	X=ob.X;
	Y=ob.Y;
	Z=ob.Z;
	cout<<"\nchiamata costruttore di copia"<<endl;
	cont++;
	cout<<cont<<endl;
}

tridim::tridim(int A, int B, int C)
{
	X=A;
	Y=B;
	Z=C;
	cout<<"\ncostruttore parametrizzato"<<endl;
	cont++;
	cout<<cont<<endl;
}

tridim::tridim()
{
	cout<<"\ncostruttore"<<endl;
	X=0;
	Y=0;
	Z=0;
	cont++;
	cout<<cont<<endl;
}

tridim::~tridim()
{
	cout<<"\ndistruttore"<<endl;
	cont--;
	cout<<cont<<endl;
}

void tridim::visualizza()
{
	cout<<"\nvalore di X: "<<X<<endl;
```

```
        cout<<"\nvalore di Y: "<<Y<<endl;
        cout<<"\nvalore di Z: "<<Z<<endl;
    }


    //main.cpp
    #include "tridim.h"

    #include <iostream>
    using namespace std;

    int tridim::cont;

    int main()
    {
        tridim ob1(1,1,1),ob2(1,5,1),ob;
        ob1.visualizza();
        ob2.visualizza();
        ob=ob1+ob2;//vengono chiamati i costruttori di copia
        ob.visualizza();
        system("PAUSE");
        return 0;
    }
```

ESERCIZIO 18

```
#include <iostream>

#include <math.h>
using namespace std;
#define pi 3.14159265

//using namespace std;

class punto {

   double prima, seconda;

public:

   punto() {} // necessario per costruire valori temporanei
   punto(double x1, double x2) {
   prima = x1;

   seconda = x2;

}

~punto() {cout <<"distruzione"<<endl;}

void show() {

   cout <<"("<< prima << " , " << seconda <<")"<< endl;

}
```

```
friend punto operator+(punto op1, punto op2);

//punto operator+(punto op2);

};

// overloading di + per punto con friend
punto operator+(punto op1, punto op2)
{

    punto temp;

    temp.prima = op2.prima*cos(pi*op2.seconda/180) + op1.prima ;

    temp.seconda = op2.prima*sin(pi*op2.seconda/180) +
    op1.seconda;

    return temp;

}

/*

 // Overloading di + per punto punto punto::operator+(punto op2)
 {

 punto temp;

 temp.prima = op2.prima*cos(pi*op2.seconda/180) + prima; temp.seconda =
op2.prima*sin(pi*op2.seconda/180) + seconda; return temp;
 }

 */

int main()

{

    double ascissa, ordinata, modulo, fase;

    cout<<"ascissa primo punto = ";
    cin >>ascissa;

    cout<<"ordinata primo punto = ";









    cin >>ordinata;

    cout<<"modulo secondo punto = ";
```

```
    cin >>modulo;

    cout<<"fase secondo punto = ";

    cin >>fase;

    punto punto_cartesiano(ascissa, ordinata); cout<<endl<<"punto cartesiano = ";
punto_cartesiano.show();
    punto punto_polare(modulo,fase); cout<<endl<<"punto polare = "; punto_polare.show();
    cout<<endl;

    punto_cartesiano=punto_cartesiano+punto_polare; cout<<endl<<"punto cartesiano
risultato= "; punto_cartesiano.show();
    cout<<endl;

    return 0;

}
```

```
/*
ascissa primo punto = 2
ordinata primo punto = 3
modulo secondo punto = 1
fase secondo punto = 90
punto cartesiano = (2 , 3)
punto polare = (1 , 90)
distruzione
distruzione
distruzione
punto cartesiano risultato= (2 , 4)
distruzione
```

```
distruzione

Press any key to continue

*/
```

COMPOSIZIONE

Serve a far comunicare + oggetti se un oggetto è contenuto in un altro.
Es:
persona e automobile
Sono entrambi classi indipendenti ma una persona possiede,"contiene", un auto.
La classe persona ha tra i membri privati uno di classe automobile(devo però dichiarare prima la classe automobile altrimenti non la riconosce).
Classe automobile è una classe contenuta.
Classe persona è classe contenitore.
Il concetto è quello di chiamare la classe contenuta tramite la classe contenitore.

ESERCIZIO 19

```
//persona.h

#include "autovettura.h"

#ifndef PERSONA_H
#define PERSONA_H

class persona{
private:
      int *eta;
      char *nome; char
      *cognome; autovettura
      macchina;
public:
      persona(int, int);
      void ins_n();
      void ins_c();
      void ins_e();
      void stampa_n();
      void stampa_c();
      void stampa_e();
      ~persona();
};
#endif
```

```cpp
//persona.cpp
#include "persona.h"

#include <iostream>
using namespace std;

persona::persona(int A=0, int B=0)
{
	eta=new int;
	cognome=new char[A];
	nome=new char[B];
	macchina.ins_m();
	//inizializzo anche l'oggetto contenuto
}
void persona::ins_n(){
	cout<<"\n inserire nome \n";
	cin>>nome;
}
void persona::ins_c(){
	cout<<"\n inserire cognome \n";
	cin>>cognome;
}
void persona::ins_e(){
	cout<<"\n inserire eta' \n";
	cin>>*eta;
}
void persona::stampa_n(){
	cout<<"\n nome: "<<nome<<endl;
}
void persona::stampa_c(){
	cout<<"\n cognome: "<<cognome<<endl;
}
void persona::stampa_e(){
	cout<<"\n eta': "<<*eta<<endl;
	cout<<"\n composizione \n";
	macchina.stampa_m(); //richiamo la composizione
}
persona::~persona(){
	macchina.~autovettura();
	delete []cognome;
	delete []nome;
	delete eta;
}
```

```
//autovetture.h
#ifndef AUTOVETTURA_H
#define AUTOVETTURA_H

class autovettura{
private:
	char *modello;
	int *cil;
public: autovettura();
	void ins_m();
	void stampa_m();
	~autovettura();
};
#endif

//autovettura.cpp
#include "autovettura.h"

#include <iostream>
using namespace std;

autovettura::autovettura()
{
	//questo costruttore viene chiamato quando viene
chiamato quello di persona
	//PRIMA DI PERSONA VIENE ESEGUITO QUESTO COSTRUTTORE
	cout<<"\n inserire dimensione nome modello\n";
	int C=0;
	cin>>C;
	modello=new char[C];
	cil=new int;
}

void autovettura::ins_m(){
	cout<<"\n Inserire il modello della macchina:\n";
	cin>>modello;
	cout<<"\ne cilindrata: "<<endl;
	cin>>*cil;
}
void autovettura::stampa_m(){//l'utilizzo dell'operatore this
è opzionale
	cout<<"\n modello"<<this->modello;
	cout<<"\n cilindrata"<<*cil;
}
autovettura::~autovettura()
```

```
{
	delete cil;
	delete []modello;
}

//main.cpp
#include "autovettura.h" //prima questo sennò il compilatore
dà errore
#include "persona.h"

#include <iostream>
using namespace std;

int main(){
	int n,c;
	cout<<"\n inserire dimensione cognome \n";
	cin>>n;
	cout<<"\n inserire dimensione nome \n";
	cin>>c;
	persona P(n,c);
	P.ins_c();
	P.ins_n();
	P.ins_e();
	P.stampa_c();
	P.stampa_n();
	P.stampa_e();
	system("PAUSE");
	return 0;
}
```