Lezione 9

Template
Ereditarietà

Introduzione ai Template

- In C++ e' possibile definire delle funzioni o classi dette *generiche* ovvero che abbiano come *parametro* il *tipo* di dato
- una funzione (classe) generica definisce una serie di operazioni applicabili ad un qualsiasi tipo di dato
- ovvero l'algoritmo implementato si applica a qualsiasi tipo
- una funzione (classe) generica si chiama funzione (classe) template

Sintassi e uso dei Template

- Sintassi:

```
template<class T> retType NomeFunzione(T);
```

- Uso con dichiarazione esplicita di tipo:

```
int main(){
int a=2;
cout<<NomeFunzione<int>(a);
return 0;
}
```

- Uso con dichiarazione implicita di tipo:

```
int main(){
int a=2;
cout<<NomeFunzione(a);
return 0;
}
```

Sintassi e uso dei Template

- Esempio:

```
template<class T> void swap(T& a, T& b){
  T temp;
  temp=a;
  a=b;
  b=temp;
}
```

- Uso:

```
int a=1;int b=2;
swap(a,b);
char c_a='x';char c_b='y';
swap(c_a,c_b);
```

Il Compilatore e i Template

- Il compilatore genera tutte le istanze utili della funzione template
- Il procedimento è equivalente ad un overloading automatico
- Nota: Vengono generate tutte le istanze che servono, che vengono poi effettivamente utilizzate nel codice esecutivo

Genericita' dei Template

- Le funzioni template sono limitate rispetto al caso generale di overloading perché non si possono specificare comportamenti diversi della funzione al variare del tipo

Esempio

```
#include <iostream>

void f(int i){cout<<“il valore è: ”<<i;}
void f(char i){cout<<“il carattere è: ”<<i;}
template<class T> void g(T i){cout<<“val: ”<<i;}

int main(){
  int a=56;
  char b=‘x’;

  f(a);//Stampa: il valore è: 56
  f(b);//Stampa: il carattere è: x

  g(a);//Stampa: val: 56
  g(b);//Stampa: val: x

  return 0;
}
```

Funzioni con più di un tipo generico

- Nel caso in cui si debbano specificare più di due tipi so si fa con la seguente sintassi:

- Sintassi

```
template<class T1, class T2> retType F(T1,T2);
```

- Es:

```
template<class T1, class T2>
void func(T1 x, T2 y){
      cout<<“prima:”<<x<<“ poi:”<<y;
}
```

Deduzione automatica del tipo

- Quando viene utilizzata una funzione template è generalmente possibile per il compilatore dedurre automaticamente il tipo

```
template<class T> retType NomeFunzione(argType (T));
```

- altre volte è impossibile:

```
template<class T1,class T2> retType NomeFunzione(T1,T2=0);
```

- in questo ultimo caso infatti si potrebbe avere ambiguità:

```
template<class T1,class T2> void f(T1,T2=0);
void main(){
   int a,b;
   float c,d;

   f(a,b);    f(c,d); f(a,d);
  f(a); //caso ambiguo
  f<int,float>(a); //ok
}
```

Overloading esplicito

- Se si esegue un overloading esplicito di una funzione template questa maschera quella generata implicitamente

```
template<class T> void g(T i){cout<<“val: ”<<i;}
void g(char i){cout<<“il carattere è: ”<<i;}

int main(){
  int a=56;
  char b=‘x’;
  g(a);//Stampa: il valore è: 56
  g(b);//Stampa: il carattere è: x
   return 0;
}
```

Uso delle funzioni generiche

```
template<class T> T max(T *v, int size){
  T max=v[0];
    for(int i=1;i<size;i++)
          if(max<v[i]) max=v[i];
    return max;
}

main(){
    int arrayI[7]={4,7,2,4,9,3,2};
    double arrayD[6]={7.1,9.4,2.6,5.7,4.8,6.9}
    int resI;
    double resD;
    resI=max(arrayI,7);
    resD=max(arrayD,6);
}
```

Classi Template

- Una classe generica o template può definire i propri membri in modo generico
- Sintassi in dichiarazione:

```
template<class T> class NomeClasse{};
```

- Sintassi in definizione:

```
template<class T> retType NomeClasse<T>::funcName(argType parameter){}
```

- Sintassi in uso:

```
main(){
      NomeClasse<type> obj(init);
}
```

Deduzione per Classi Template

- Per le classi si deve sempre esplicitare il tipo nella dichiarazione
- Non ci sono meccanismi di deduzione automatica

Esempio

```
//Vettore generico
template<class T> class Vector{
public:
    Vector(int usr_size=10){size=usr_size; v=new T[size];}
    ~Vector(){delete [] v;}
    T& operator[](int);
private:
    int size;
    T* v;
};

template<class T> T& Vector<T>::operator[](int i){
    return v[i];
}
```

Esempio

```
void main{
    Vector<int> vI(100);
    Vector<char> vC(5);

    int i;
    for(i=0;i<100;i++)
          vI[i]=i*i;

    for(i=0;i<5;i++)
          vC[i]='e';

    for(i=0;i<100;i++)
          cout<<vI[i]<<" ";
    cout<<endl;
}
```

Esempio Stack

```
#ifndef TSTACK1_H
#define TSTACK1_H
template< class T >
class Stack {
public:
   Stack( int = 10 );  // default constructor (stack size 10)
    ~Stack(){delete [] stackPtr;} // destructor
   bool push( const T& );  // push an element onto the stack
   bool pop( T& );         // pop an element off the stack
   bool isEmpty() const{return top == -1;} // determine
                                           //whether Stack is empty
    bool isFull() const{return top == size - 1;} // determine whether Stack is full
private:
   int size;     // # of elements in the stack
   int top;      // location of the top element
   T *stackPtr;  // pointer to the stack
};
```

```
// constructor
template< class T >
Stack< T >::Stack( int s ){
   size = s > 0 ? s : 10;
   top = -1;  // Stack initially empty
   stackPtr = new T[ size ]; // allocate memory for elements
}

// push element onto stack;
// if successful, return true; otherwise, return false
template< class T >
bool Stack< T >::push( const T &pushValue ){
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue;  // place item on Stack
      return true;  // push successful
   } // end if
   return false;  // push unsuccessful
}
```

```
// pop element off stack;
// if successful, return true; otherwise, return false
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // remove item from Stack
      return true;  // pop successful
    } // end if
   return false;  // pop unsuccessful
}
#endif
```

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"  // Stack class template definition
int main()
{
   Stack< double > doubleStack( 5 );
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleStack\n";

   while ( doubleStack.push( doubleValue ) ) {
      cout << doubleValue << ' ';
      doubleValue += 1.1;
   }
```

```
 cout << "\nStack is full. Cannot push " << doubleValue
      << "\n\nPopping elements from doubleStack\n";

while ( doubleStack.pop( doubleValue ) )
   cout << doubleValue << ' ';

cout << "\nStack is empty. Cannot pop\n";

Stack< int > intStack;
int intValue = 1;
cout << "\nPushing elements onto intStack\n";

while ( intStack.push( intValue ) ) {
   cout << intValue << ' ';
   ++intValue;
 }
```

```
   cout << "\nStack is full. Cannot push " << intValue
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   return 0;
}
```

Ereditarietà

- L’ereditarietà è importante per la creazione di software riutilizzabile e per controllare la complessita’ del codice
- classi nuove sono progettate sulla base di classi pre-esistenti
- le nuove classi acquisiscono gli attributi e i comportamenti delle classi precedenti ed aggiungono caratteristiche nuove o raffinano caratteristiche pre-esistenti

Ereditarietà

- Quando si crea una nuova classe si può fare in modo che questa erediti i dati membro e le funzioni membro da una classe già definita precedentemente
- la classe precedente prende il nome di *classe base*
- la classe che eredita prende il nome di *classe derivata*

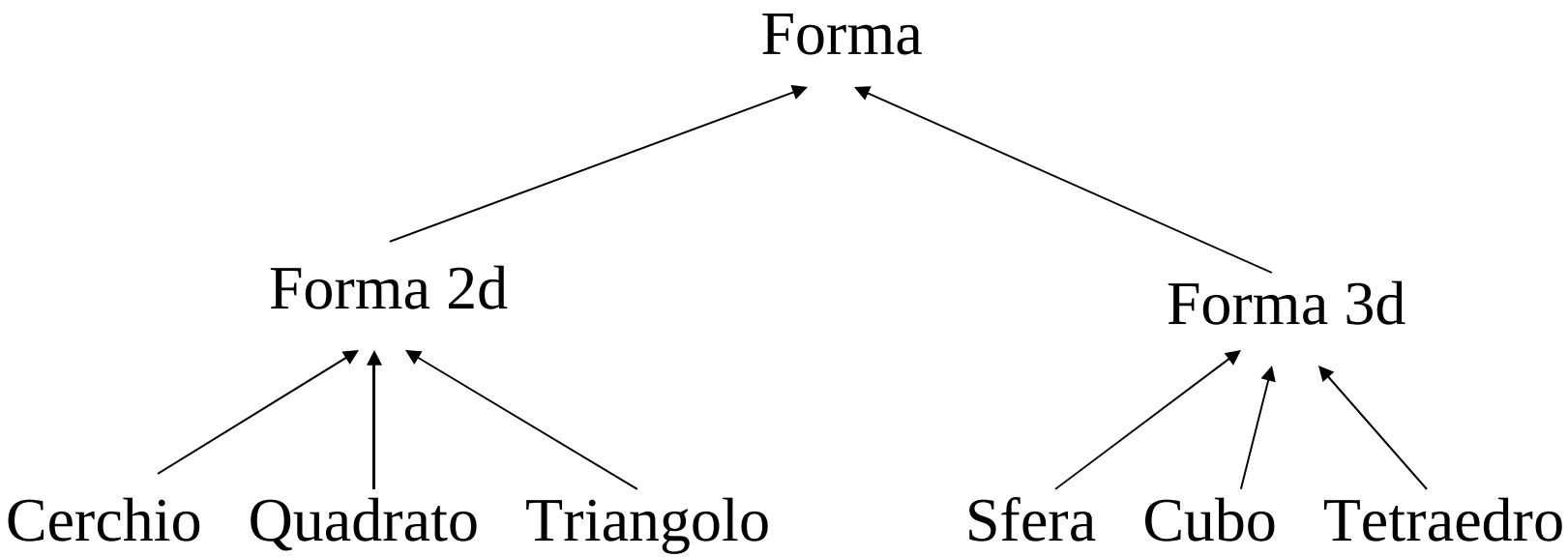
Ereditarietà gerarchica

- E’ possibile continuare il procedimento di ereditarietà creando una classe che eredita a sua volta da una classe derivata
- questo procedimento crea una gerarchia
- una classe base può essere *diretta* o *indiretta* se si trova a livelli più alti della gerarchia di derivazione
- ovvero se C eredita da B che eredita da A allora:
  - B è una classe base diretta per C
  - A è una classe base indiretta per C

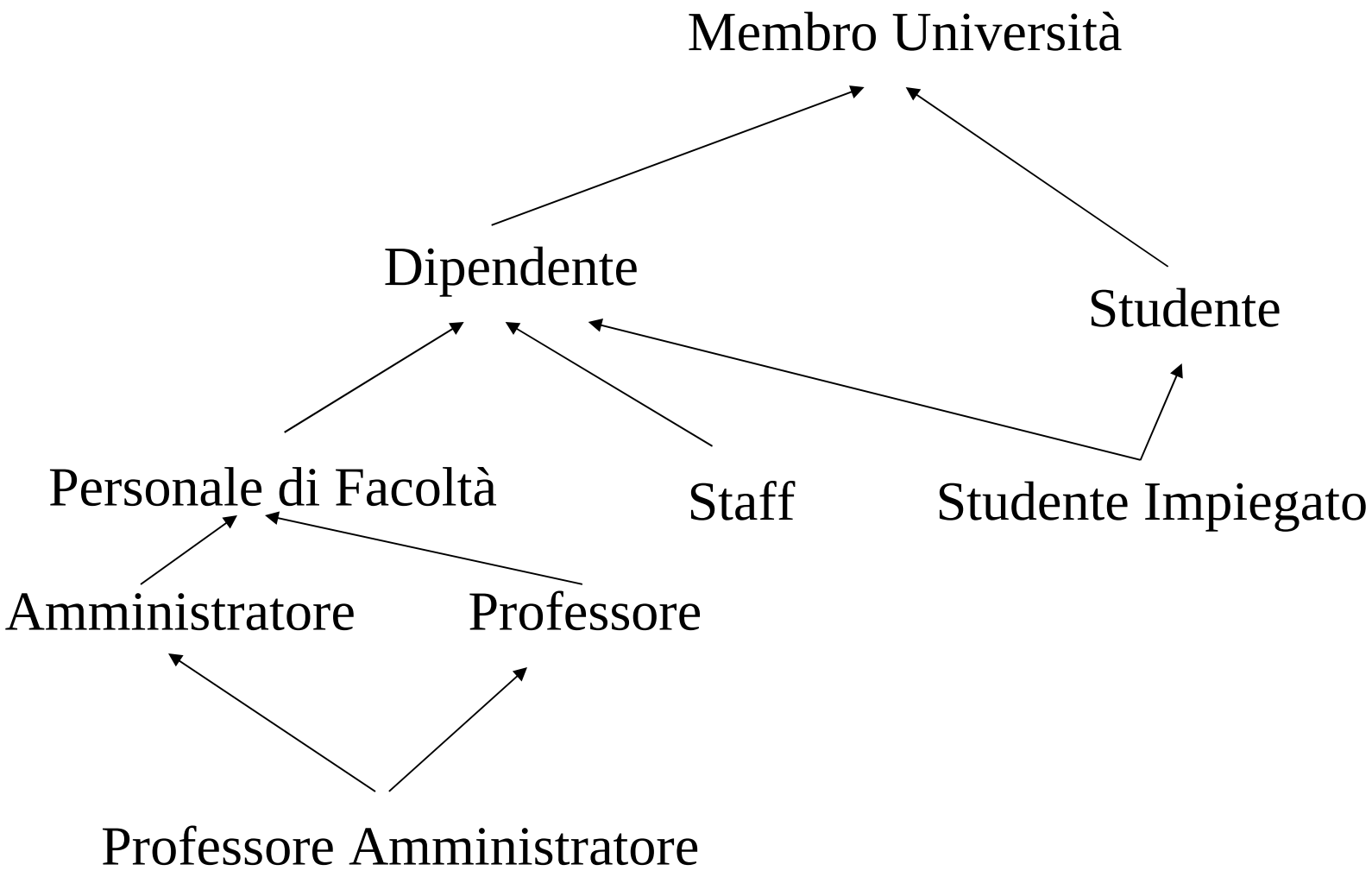
Ereditarietà

- Esistono due tipi di ereditarietà:
  - *singola*: quando una classe derivata eredita da una sola classe base
  - *multipla*: quando una classe derivata eredita da più classi base che tra loro possono non essere correlate

Ereditarietà singola



Ereditarietà multipla



Cosa può ereditare una classe?

- la classe derivata acquisisce i
  dati membro
- e le
  funzioni membro
- della classe base

La sintassi

- Nella definizione della classe derivata si aggiunge la specifica della classe base da cui si eredita ed il tipo di eredità

```
class BaseClass{
  //dichiarazione
};

class DerivedClass: public BaseClass{
  //dichiarazione
};
```

- Vedremo in seguito che esistono tre tipi di ereditarietà:

  public, private, protected

Costruttore di classe derivata

- Il costruttore della classe derivata si deve occupare di inizializzare i dati membri aggiuntivi, quelli cioè che sono introdotti in più rispetto ai dati membro della classe base
- si può utilizzare il costruttore della classe base per inizializzare i dai membri *condivisi* con la classe base
- la sintassi è:

```
NomeClassDeriv(T prm_bas,T prm_drv):NomeClassBase(prm_bas){
      //init con prm_drv
}
```

Dichiarazione di una classe base

```
#ifndef POINT_H
#define POINT_H

class Point{
    friend ostream operator<<(ostream &, const Point &);

public:
    Point(int=0, int=0);
    void setPoint(int, int);
    int getX() const {return x;}
    int getY() const {return y;}

protected:
    int x,y;
};

#endif
```

Definizione delle funzioni

```
#include<iostream>
#include “point.h”

Point::Point(int a, int b){set(a,b);}

void Point::set(int a, int b){x=a;y=b;}

ostream& operator<<(ostream &out, const Point &p){
    out<<“[”<<p.x<<“,”<<p.y<<“]”<<endl;
    return out;
}
```

Dichiarazione di una classe derivata

```
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle: public Point{
    friend ostream& operator<<(ostream &, Circle &);
public:
    Circle(double r=0.0, int x=0, int y=0);
    void setRadius(double);
    double getRadius() const {return radius;}
    double area() const;
protected:
    double radius;
}

#endif
```

Definizione delle funzioni

```
#include<iostream>
#include “circle.h”
 Circle::Circle(double r, int a, int b): Point(a,b){//Costruttore per la classe
    base
    setRadius(r);
}
void Circle::setRadius(double r){
    radius=(r>=0 ? r: 0);
}
double Circle::area()const{
    return 3.14159 * radius *radius;
}
ostream & opertor<<(ostream &out, Circle &c){
    out<<“Center:”<< static_cast<Point>( c ) //cast esplicito
    <<“Radius:”<<c.radius<<endl;
    return out;
}
```

Esempio

```
#include<iostream>
#include “point.h”
#include “circle.h”
int main(){
    Point p(10,20);
    Circle c(2.1,30,40);
    cout<<c; //Stampa: Center:[30,40] Radius:2.1
    Point *pPtr=&c;
    cout<<(*pPtr); // Stampa: [30,40] il punt vede solo
//i dati membri della classe base
    Circle *cPtr=static_cast<Circle *>(pPtr);
    cout<<(*cPtr); // Stampa: Center: [30,40] Radius:2.1.
//I dati ci sono sempre. Erano solo non visibili prima
    cPtr=static_cast<Circle *>(&p);
    cout<<(*cPtr); // Stampa: Center:[30,40] Radius:???
//Adesso invece non sono mai esistiti e quindi si va ad
//accedere in memoria casualmente. ERRORE
}
```

Note

- Il cast esplicito ha sintassi:

  `static_cast<tipo>(oggetto)`
- il suo compito è di eseguire (al tempo di compilazione) una conversione forzata al tipo indicato dell'oggetto passato come parametro
- E' possibile passare da una classe derivata alla sua classe base
- E' errato passare da una classe base ad una derivata! Infatti non esistono e non sono accessibili i dati e funzioni membro aggiunte dalla classe derivata: se si tenta di accedervi si genera un errore di violazione di memoria

Specializzazione

- la classe derivata aggiunge dati membro e funzioni membro a quelle della classe base
- la classe derivata specializza, raffina, reimplementa le funzioni membro della classe base
- una classe derivata è più *grande* di una classe base nel senso che occupa più spazio, ha più dati e funzioni membro
- una classe derivata rappresenta tuttavia un gruppo più *ristretto* di oggetti, è più specializzata

Overriding di funzioni membro

- Una classe derivata può ridefinire una funzione membro della classe base
- Attenzione non è overloading: infatti la funzione ha lo stesso nome e gli stessi parametri
- se fosse stato un caso di overloading la nuova funzione si sarebbe dovuta distinguere per qualche parametro
- la redefinizione si chiama *overriding*
- la funzione nella classe base è mascherata dalla funzione ridefinita nella classe derivata

Overriding di funzioni membro

- Una classe derivata può aver bisogno di accedere alle funzioni della classe base
- se le funzioni sono state ridefinite tramite overriding sorge il problema di indicarle senza ambiguità
- lo si può fare utilizzando l’operatore di risoluzione :: ed indicando il nome della classe base

Esempio

```
// Definition of class Employee
#ifndef EMPLOY_H
#define EMPLOY_H

class Employee {
public:
   Employee( const char *, const char * );  // constructor
   void print() const;  // output first and last name
   ~Employee();         // destructor
private:
   char *firstName;     // dynamically allocated string
   char *lastName;      // dynamically allocated string
};
#endif
```

```
// Member function definitions for class Employee
#include <iostream>
#include <cstring>
#include <cassert>
#include "employ.h”
Employee::Employee( const char *first, const char *last ){
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 ); // terminate if not allocated
   strcpy( firstName, first );
   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );  // terminate if not allocated
   strcpy( lastName, last );
}
void Employee::print() const
   { cout << firstName << ' ' << lastName; }
Employee::~Employee(){
   delete [] firstName;   // reclaim dynamic memory
   delete [] lastName;    // reclaim dynamic memory
}
```

```
// Definition of class HourlyWorker
#ifndef HOURLY_H
#define HOURLY_H

#include "employ.h"

class HourlyWorker : public Employee {
public:
   HourlyWorker( const char*, const char*, double, double );
   double getPay() const;  // calculate and return salary
   void print() const;     // overridden base-class print
private:
   double wage;            // wage per hour
   double hours;           // hours worked per week
};

#endif
```

```
// Member function definitions for class HourlyWorker
#include <iostream>
#include <iomanip>
#include "hourly.h"

HourlyWorker::HourlyWorker( const char *first,
                            const char *last,
                            double initHours, double initWage )
   : Employee( first, last )   // call base-class constructor
{
   hours = initHours;  // should validate
   wage = initWage;    // should validate
}

// Get the HourlyWorker's pay
double HourlyWorker::getPay() const { return wage * hours; }
```

```
// Print the HourlyWorker's name and pay
void HourlyWorker::print() const
{
   cout << "HourlyWorker::print() is executing\n\n";
   Employee::print();   // call base-class print function

   cout << " is an hourly worker with pay of $"
        << setiosflags( ios::fixed | ios::showpoint )
        << setprecision( 2 ) << getPay() << endl;
}
```

```
#include "hourly.h"
int main()
{
   HourlyWorker h( "Bob", "Smith", 40.0, 10.00 );
   h.print();
   return 0;
}
```

Protezione di ereditarietà

- L’eredità in C++ può essere di tre tipi:
  - pubblica (**`public`**)
  - privata (**`private`**)
  - protetta (**`protected`**)

- si distinguono in base alle restrizioni di accesso che si realizzano su i dati membro e le funzioni membro ereditate

Significato intuitivo

- Ereditare in modo pubblico significa che ciò che prima era visibile all'esterno rimane visibile all'esterno anche dopo l'eredità

- Ereditare in modo protetto o privato significa che ciò che prima era visibile all'esterno rimane visibile solo all'interno della classe che eredita

Eredità pubblica:

- ciò che è pubblico nella classe base *è* accessibile alla classe derivata ed *è* accessibile all'esterno
- ciò che è protetto nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno
- ciò che è privato nella classe base *non è* accessibile alla classe derivata e *non è* accessibile all'esterno

Eredità pubblica:

```
Class Base{
public:
   Base(int usr_a=0, int usr_b=0, int usr_c=0)
   {a_pub=usr_a;b_pro=usr_b;c_pri=usr_c;}
   int a_pub;
protected:
   int b_pro;
private:
   int c_pri;
};
```

Eredità pubblica:

```
class Derivata: public Base{
public:
   Derivata(int usr_a=0, int usr_b=0, int usr_c=0, int
   usr_a_d=0, int usr_c_d=0)
   :Base(usr_a,usr_b,usr_c){a_d=usr_a_d;c_d=usr_c_d;}
   int get_a(){return a_pub;}
   int get_b(){return b_pro;}
   int get_c(){return c_pri;}
   int a_d_pub;
private:
   int c_d_pri;
};
```

Eredità pubblica:

```
Void main(){
   Derivata obj(1,2,3,4,5);
   //visibilità dal main
   obj.a_pub=10;//OK
   obj.b_pro=10;//NO
   obj.c_pri=10;//NO
   obj.a_d_pub=10;//OK
   obj.c_d_pri=10;//NO

   //visibilità nella classe derivata
   obj.get_a();//OK
   obj.get_b();//OK
   obj.get_c();//NO la classe derivata non ha accesso al dato
   privato della classe base
}
```

Eredità protetta:

- ciò che è pubblico nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno

  (ma può essere tramandato ai discendenti in modo che questi possano accedervi)
- ciò che è protetto nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno

- ciò che è privato nella classe base *non è* accessibile alla classe derivata e *non è* accessibile all'esterno

Eredità protetta:

```
Void main(){
   Derivata obj(1,2,3,4,5);

   obj.a=10;//NO
   obj.b=10;//NO
   obj.c=10;//NO
   obj.a_d=10;//OK
   obj.c_d=10;//NO

   obj.get_a();//OK
   obj.get_b();//OK
   obj.get_c();//NO
}
```

Eredità privata:

- ciò che è pubblico nella classe base *è* accessibile alla classe derivata e *non è* accessibile all'esterno

  (né è accessibile ai discendenti)
- ciò che è protetto nella classe base *è* accessibile alla classe derivata ma *non è* accessibile all'esterno

- ciò che è privato nella classe base *non è* accessibile alla classe derivata e *non è* accessibile all'esterno

Eredità privata:

```
Void main(){
   Derivata obj(1,2,3,4,5);

   obj.a=10;//NO
   obj.b=10;//NO
   obj.c=10;//NO
   obj.a_d=10;//OK
   obj.c_d=10;//NO

   obj.get_a();//OK
   obj.get_b();//OK
   obj.get_c();//NO
}
```

Eredità protetta e privata:

- i membri pubblici e protetti della classe base se ereditati in modo protetto diventano protetti nella classe derivata
- i membri pubblici e protetti della classe base se ereditati in modo privato diventano privati nella classe derivata

Eredità protetta e privata:

- La differenza fra l'ereditarietà protetta o privata è rilevabile solo se la classe derivata è a sua volta ereditata da una o più classi gerarchicamente
- nel caso di eredità protetta i membri pubblici e protetti rimangono accessibili per tutta la gerarchia (ma non all'esterno)
- mentre nel caso di eredità privata i membri di qualsiasi tipo cessano di essere trasmessi in modo accessibile ai discendenti

Eredità privata:

- Esempio di utilità del meccanismo di ereditarietà privata
  - data una classe ListClass che manipola le liste si può derivare una classe QueueClass che la eredita in modalità privata
  - tutti i metodi di ListClass diventano privati di QueueClass e non accessibili all'esterno
  - si definiscono i metodi pubblici di QueueClass che non fanno altro che richiamare i metodi di ListClass di interesse
    - ex: enqueue chiama insertAtBack e dequeue chiama removeFromFront
  - gli altri metodi rimangono inaccessibili

Costruttori e distruttori in classi derivate

- Dato che una classe derivata contiene i membri della classe base quando viene istanziata deve poter accedere al costruttore della classe base

- Sintassi:

```
ClasseDerivata(arg_base, arg_deriv):ClasseBase(arg_base){...}
```

- Se non viene fatto in modo esplicito allora la classe derivata chiama in modo implicito il costruttore di default della classe base

Costruttori e distruttori in classi derivate

- Dato che il distruttore viene invocato automaticamente e non prende parametri, la classe derivata non ha un modo esplicito per invocare il distruttore della classe base

Ordine di chiamata dei Costr/Distr

```
#include<iostream>
class Base{
public:
   Base(){cout<<“Costruzione Base\n”;}
   ~Base(){cout<<“Distruzione Base\n”;}
};
class Deriv1:public Base{
public:
   Deriv1(){cout<<“Costruzione Deriv1\n”;}
   ~Deriv1(){cout<<“Distruzione Deriv1\n”;}
};
class Deriv2:public Deriv1{
public:
   Deriv2(){cout<<“Costruzione Deriv2\n”;}
   ~Deriv2(){cout<<“Distruzione Deriv2\n”;}
};
```

```
void main(){
   Deriv2 ob;
}
```

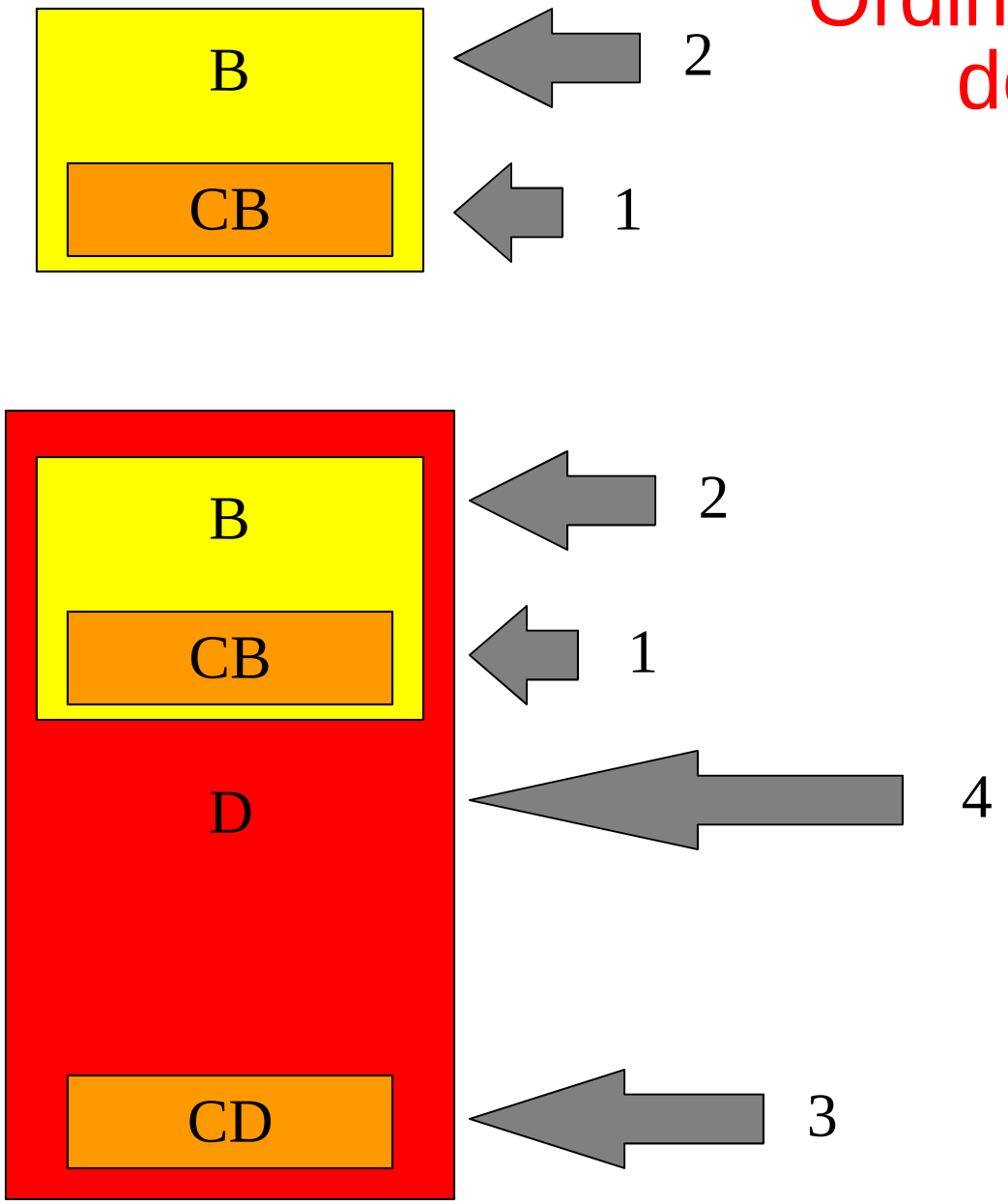
Output:

```
Costruzione Base
Costruzione Deriv1
Costruzione Deriv2
Distruzione Deriv2
Distruzione Deriv1
Distruzione Base
```

Nota sui costruttori

- Si ha un oggetto di una classe derivata D da una classe base B. Sia B che D contengono oggetti di altre classi CB e CD rispettivamente
- quando si crea un oggetto di tipo D sono eseguiti prima i costruttori degli oggetti CB poi il costruttore di B, poi i costruttori degli oggetti CD e infine il costruttore di D
- I distruttori sono chiamati in ordine inverso rispetto ai corrispondenti costruttori

Ordine di invocazione dei costruttori



Eredità singola o multipla

- Ereditarietà singola: la classe derivata eredita solo da una classe base
- Ereditarietà multipla: la classe derivata eredita da più classi base (anche fra di loro eterogenee)

Eredità multipla

```
class Base1{
public:
    Base1(int i=0){b1=i;}
private:
    int b1;
};

class Base2{
public:
    Base2(int i=0){b2=i;}
private:
    int b2;
};

class Deriv:public Base1, public Base2{
public:
    Deriv(int i=0, int j=0, int z=0):Base1(i), Base2(j){d=z;}
private:
    int d;
};
```

Relazioni fra classi

- Due classi possono essere in relazione l'una con l'altra nei modi:
  - **è un**
  - **ha un**

Relazione: è un

- definizione: una classe è una specializzazione di una seconda classe
- implementazione: tramite il meccanismo di ereditarietà
- Es: una classe cerchio è una classe punto (a cui si è aggiunto dati e membri)

Relazione: ha un

- definizione: una classe ha nella sua composizione altre classi
- implementazione: una classe ha fra i suoi dati membro altre classi
- Es: una classe Impiegato ha fra i propri attributi la classe Anagrafica e la classe Azienda