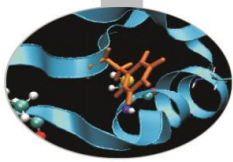
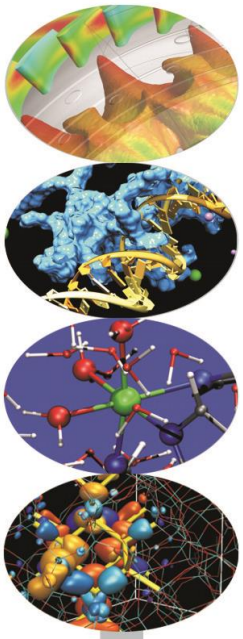
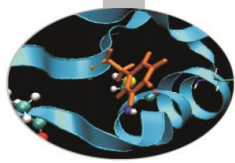


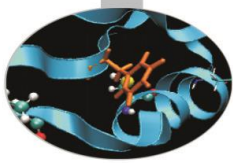
Ereditarietà





Indice

- **L'ereditarietà**
- **L'ereditarietà di tipo public**
- **Costruttori e distruttori**
- **Ereditarietà e composizione**
- **L'ereditarietà di tipo private e protected**
- **L'overriding**
- **L'ereditarietà multipla**
- **Casting**



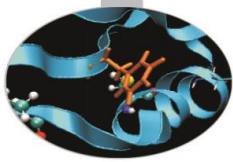
Ereditarietà

- L'***ereditarietà*** è lo strumento che permette di costruire nuove classi, dette ***classi derivate***, a partire da classi già esistenti (***classi base***) favorendo in tal modo il riutilizzo del codice.
- In generale, quando viene stabilita una relazione di ereditarietà fra due classi, la classe derivata conterrà in sé i dati e le funzioni membro della classe base. Alcuni metodi, tuttavia, non possono essere ereditati automaticamente.
- Parliamo di ereditarietà ***singola***, quando la classe base è unica, e di ereditarietà ***multipla*** quando, al contrario, una classe derivata è costruita a partire da almeno due classi base.
- Possiamo distinguere tra ereditarietà ***diretta*** o ***indiretta*** tra due classi. Nel primo caso una classe derivata eredita il contenuto della classe base senza passaggi intermedi:

  classe base *classe derivata*

- Nel secondo caso, invece, si è in presenza di un gerarchia di classi:

  classe base *classe derivata1 classe derivata2*



Ereditarietà

- Da un punto di vista sintattico, nell'ereditarietà diretta il nome della classe base compare esplicitamente nella porzione di codice relativa alla classe derivata.

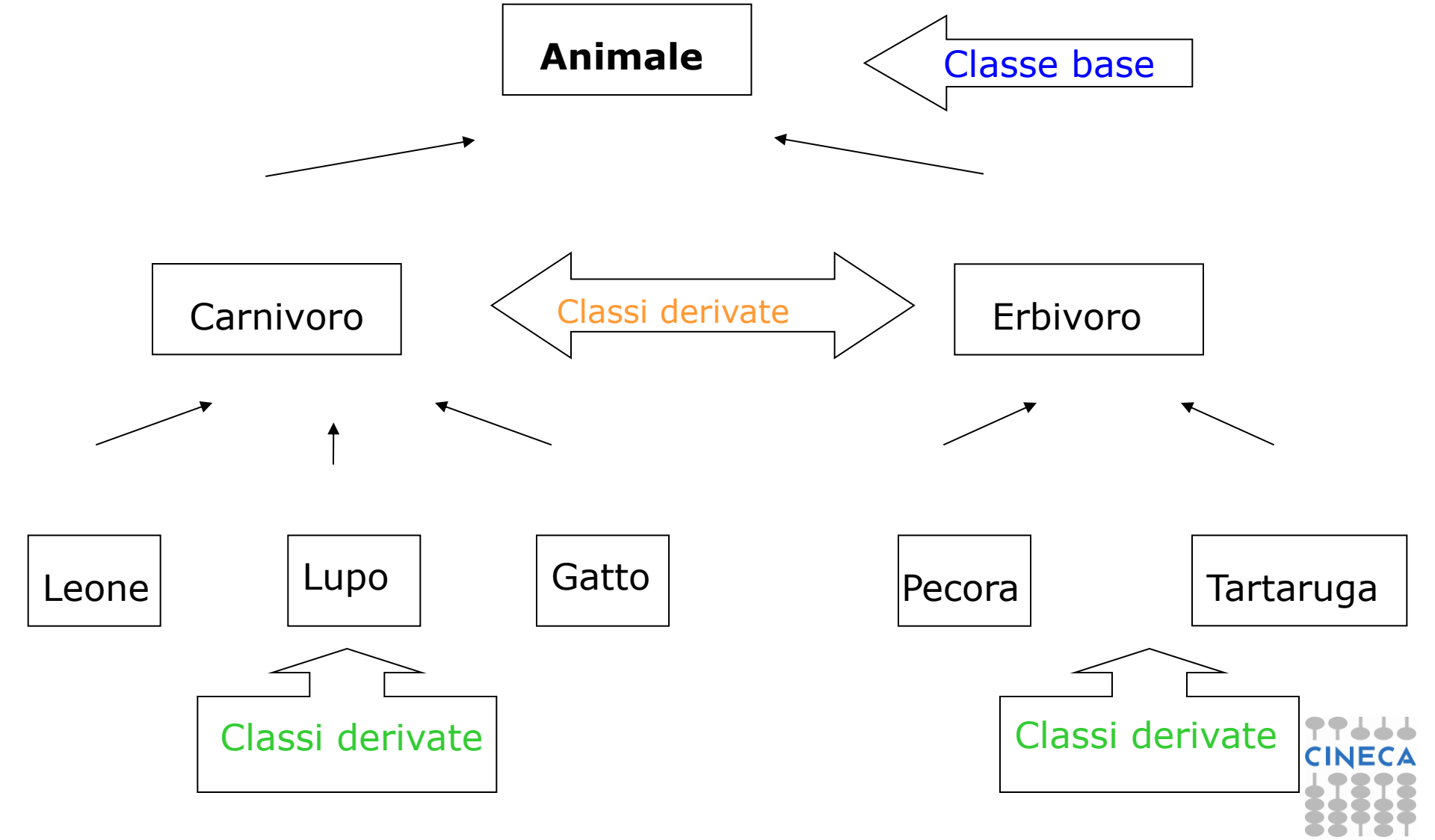
- Il C++, inoltre, mette a disposizione tre diverse tipologie di ereditarietà: ***public***, ***protected*** e ***private***. L'ereditarietà di tipo *public* è di gran lunga la più usata.

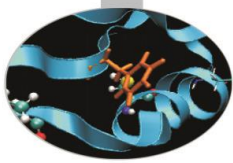
- Nell'ereditarietà di tipo public, un oggetto di una classe derivata può essere visto come un'istanza particolare della classe base.

  Non è vero, invece, il contrario.

- L'ereditarietà fornisce un'alternativa alla *composizione* dove un oggetto di una classe viene trattato come dato membro di un'altra classe. Il risultato che si ottiene con queste due diverse modalità di programmazione è sostanzialmente lo stesso.

Ereditarietà



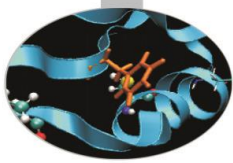


Ereditarietà

Animale
attributi:
nome
età
colore
metodi:
svegliarsi()
muoversi()
fermarsi()
nutrirsi()
dormire()
setNome()…
getNome()…
Animale()

is-a

Carnivoro
attributi:
prede
numeroDenti
metodi:
cacciare()
setPrede()…
getPrede()…
Carnivoro()



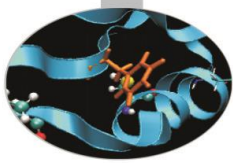
Ereditarietà

Carnivoro
attributi:
nome
età
colore
prede
numeroDenti
metodi:
svegliarsi()
muoversi()
fermarsi()
nutrirsi()
dormire()
setNome()
getNome()
cacciare()
setPrede()
getPrede()
Carnivoro()

is-a

Gatto
attributi:
razza
metodi:
miagolare()
setRazza()
getRazza()
Gatto()

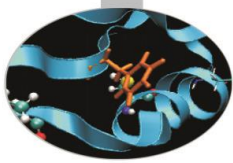
Ereditarietà



Gatto
attributi:
nome
età
colore
prede
numeroDenti
razza
metodi:
svegliarsi()
muoversi()
fermarsi()
nutrirsi()
dormire()
setNome()…

getNome()…
cacciare()
setPrede()…
getPrede()…
miagolare()
setRazza()
getRazza()
Gatto()

L'ereditarietà di tipo public

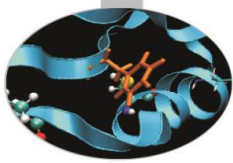


- La dichiarazione di ereditarietà di tipo public si avvale della seguente sintassi:

```
class nome_classe_derivata : public  nome_classe_base{
   corpo della classe derivata;
};
```

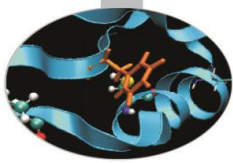
- Nell'eredità di tipo public, la classe derivata vede i membri ***public*** e ***protected*** della classe base ancora come ***public*** e ***protected***: essi restano direttamente accessibili rispettivamente dall'esterno e dalle altre classi derivate (basta cioè invocarne il nome).
- I membri ***private*** della classe base possono essere acceduti solo attraverso i metodi public e protected della classe base stessa: ***non*** sono direttamente accessibili dalla classe derivata.

L'ereditarietà di tipo public

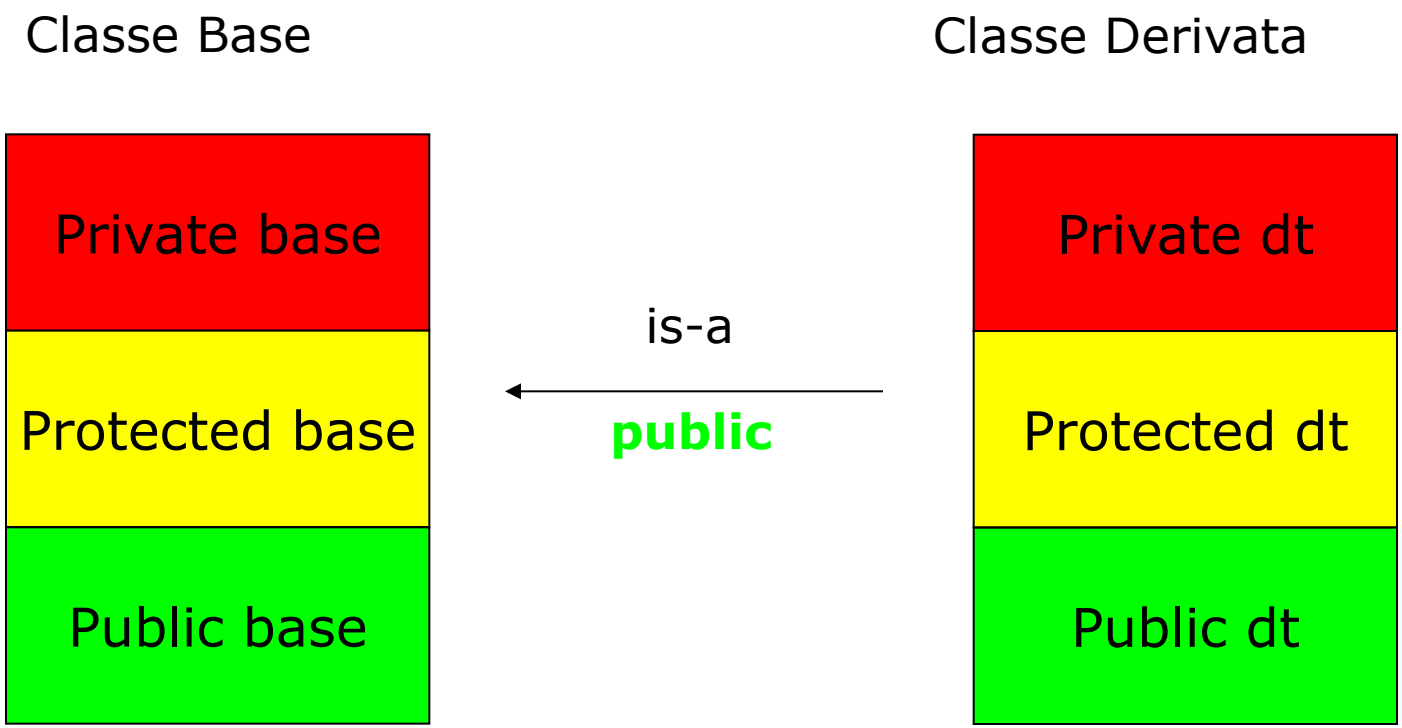


Le funzioni ***friend*** della classe base ***non*** sono ereditate, essendo definite al di fuori della classe base.

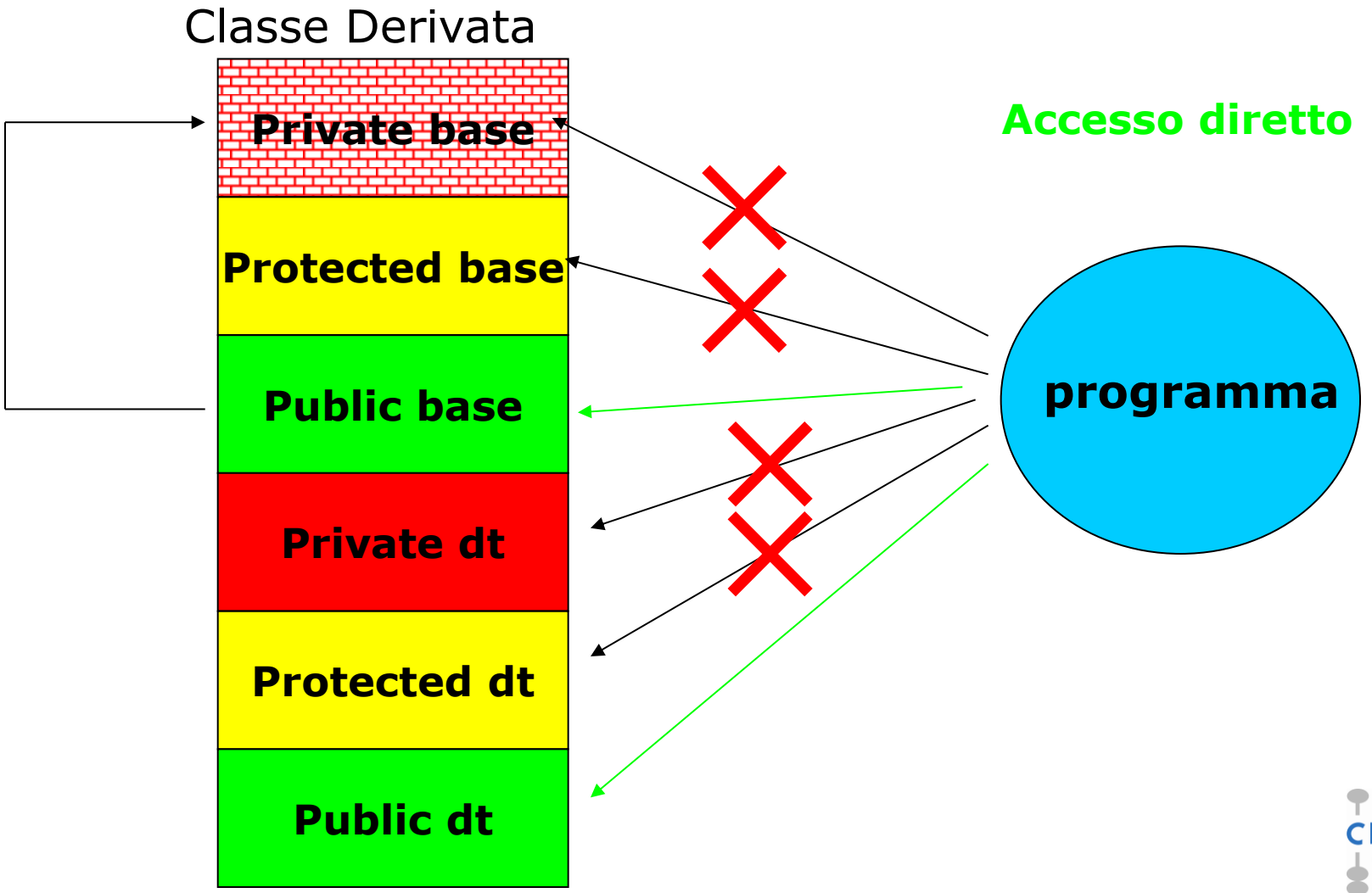
- Con l'attributo ***protected*** si identificano quei membri della classe base che sono direttamente accessibili dalle classi derivate, ma non dall'esterno (in questo caso bisognerà ricorrere a metodi public o funzioni friend della classe base).
- L'ereditarietà di tipo public permette di trattare un oggetto della classe derivata come un particolare oggetto della classe base.



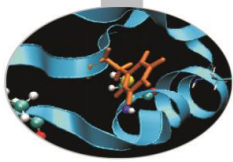
L'ereditarietà di tipo public

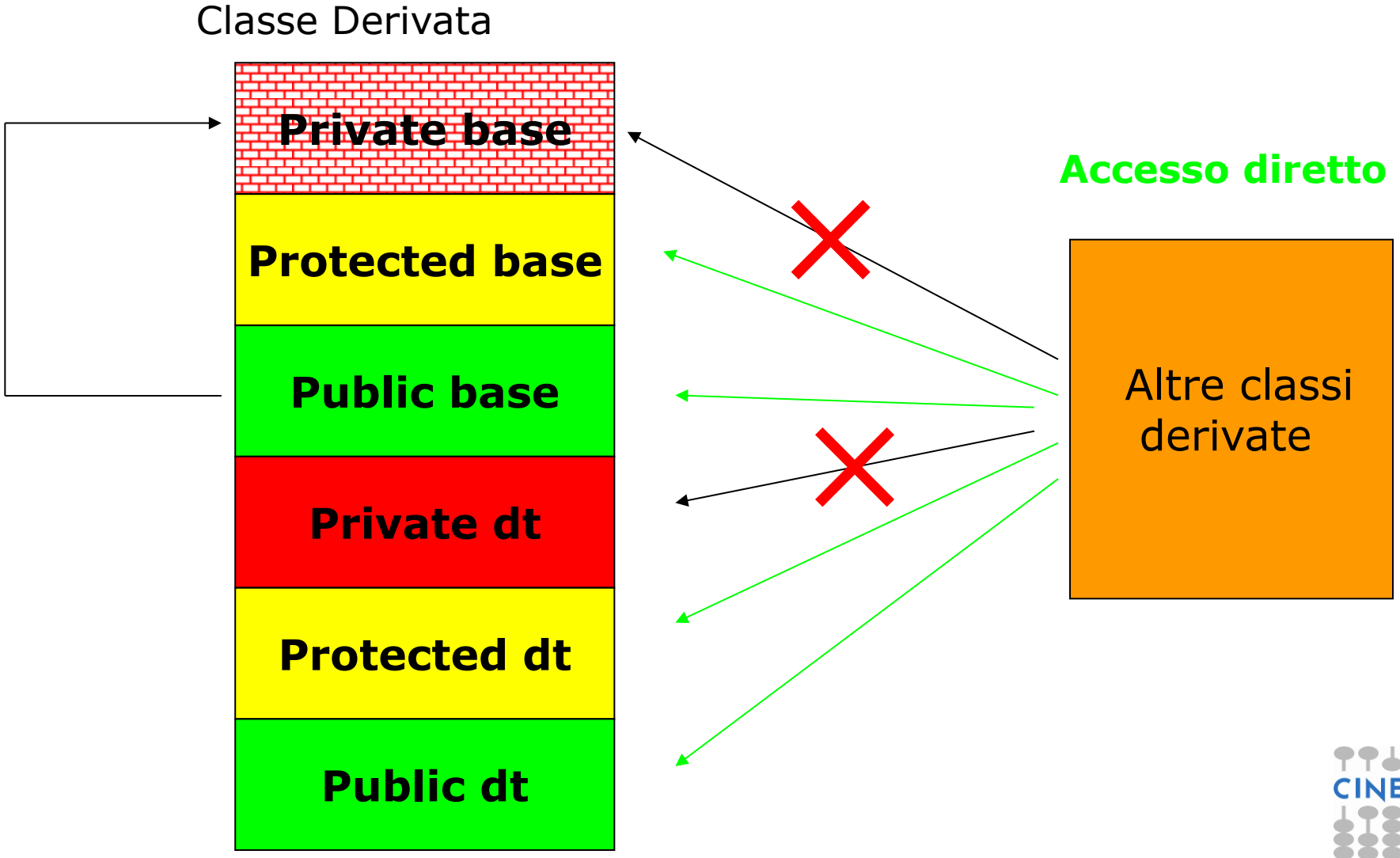


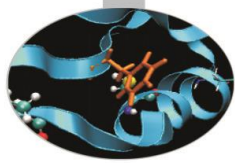
L'ereditarietà di tipo public



L'ereditarietà di tipo public



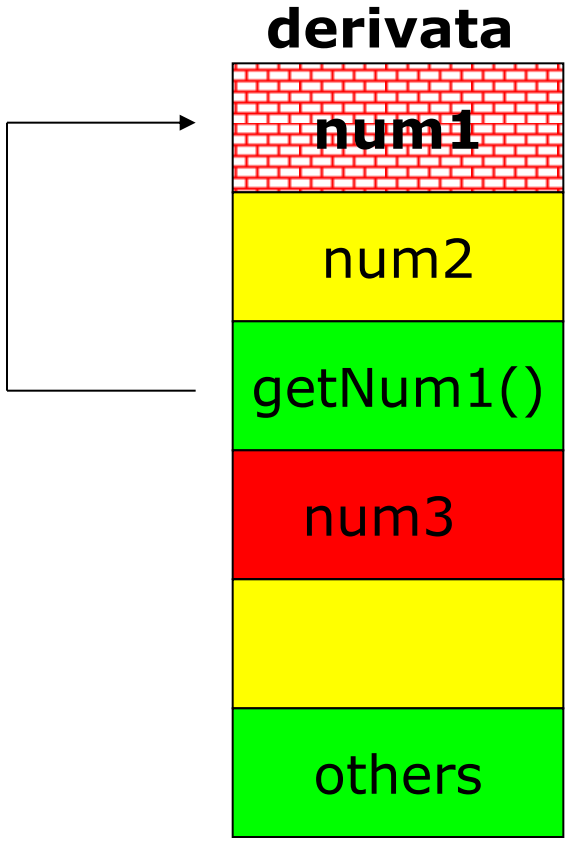


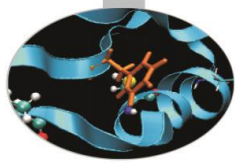


Esempio

- esempio: sintassi nell'ereditarietà di tipo public

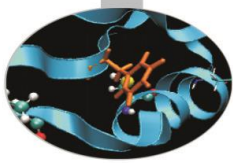
```
// file base.h
class Base{
   private:
      int num1;
   protected:
      int num2;
   public:
      Base(){num1=0; num2=0;}
      void setNum1(int i){ num1=i; }
      void setNum2(int i){ num2=i; }
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
};


sum;
      }
};
```





Esempio

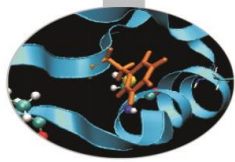
```
// file derivata.h
#include "base.h"

class Derivata: public Base{
    private:
      int num3;
    public:
      Derivata(){ num3=0; }
      void setNum3(int i){ num3=i;}
      int getNum3(){ return num3;}
      int somma(){
        int sum;
        sum=num3+num2+getNum1();
        return
}
```



Esempio

```
// file main.cpp

#include<iostream>
#include "derivata.h"
using namespace std;
int main(){
    Base b;
    Derivata d;
    b.setNum1(1); b.setNum2(2);
    d.setNum1(4); d.setNum2(5); d.setNum3(6);
    cout << "Size of b: " << sizeof(b) << endl;
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "b->Num1: " << b.getNum1() << " Num2: "
         << b.getNum2() << endl;
    cout << "d->Num1: " << d.getNum1() << " Num2: " <<
   d.getNum2()
         << " Num3: " << d.getNum3() << endl;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

Esempio



Output:

Size of b: **8** ***(4 byte * 2 int)***

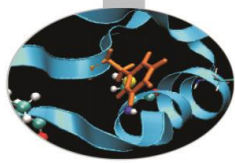
Size of d: **12** ***(4 byte * 3 int)***

b->Num1: 1 Num2: 2

d->Num1: 4 Num2: 5 Num3: 6

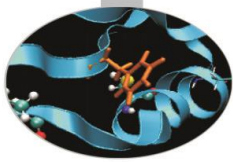
Sum: 15

Costruttori e Distruttori nelle classi derivate

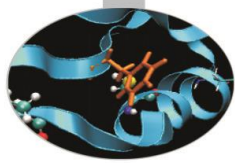


- In mancanza dell'inizializzatore di membro, viene chiamato implicitamente il costruttore di default della classe base.
- Una classe derivata eredita gli attributi della classe base, quindi, quando viene istanziato un oggetto della classe derivata è necessario chiamare il costruttore della classe base. Questa operazione viene eseguita tramite un ***inizializzatore della classe base***, realizzato secondo la sintassi:
- nome_classe_derivata(argomenti_ereditati,argomenti_propri) : nome_classe_base(argomenti_ereditati);
- Nel prototipo del costruttore della classe derivata gli argomenti ereditati possono anche seguire gli argomenti propri.

Costruttori e Distruttori nelle classi derivate



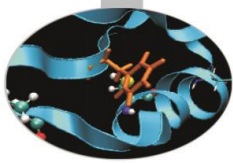
- I costruttori e gli operatori di assegnamento della classe base **non sono ereditati** dalle classi derivate ma possono essere invocati, rispettivamente, dai costruttori e dagli operatori di assegnamento delle classi derivate stesse.
- Quando vengono creati oggetti di una classe derivata, per primi sono chiamati i costruttori della classe base, poi quelli della classe derivata. I distruttori sono chiamati in ordine inverso.



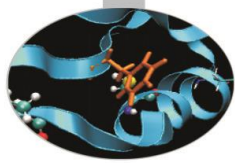
Esempio

```
// file base.h
#include<iostream>
using namespace std;
class Base{
   private:
      int num1;
   protected:
      int num2;
   public:
      Base(int i1, int i2){
        num1=i1; num2=i2;
        cout << "Costruttore Base: " << num1 << " " << num2
             << endl;}
      ~Base(){ cout << "Distruttore Base: " << num1 << " "
                    << num2 << endl;}
      void setNum1(int i){ num1=i; }
      void setNum2(int i){ num2=i; }
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
};
```

Esempio



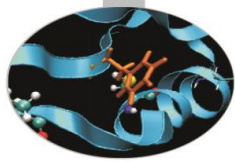
```
// file derivata.h
#include "base.h"

class Derivata: public Base{
    private:
      int num3;
    public:
      Derivata(int i1, int i2, int i3):Base(i1,i2){
        num3=i3;
        cout << "Costruttore Derivata: " << num3 << endl;
      }
      ~Derivata(){ cout << "Distruttore Derivata: " << num3
                        << endl;
      }
      void setNum3(int i){ num3=i;}
      int getNum3(){ return num3;}
      int somma(){
        int sum;
        sum=num3+num2+getNum1();
        return sum;
      }};
```



Esempio

```
// file main.cpp
#include "derivata.h"

int main(){
    Base b(4,5);
    Derivata d(b.getNum1(),b.getNum2(),6);
    cout << "Size of b: " << sizeof(b) << endl;
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "Num1: " << d.getNum1() << endl;
    cout << "Num2: " << d.getNum2() << endl;
    cout << "Num3: " << d.getNum3() << endl;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

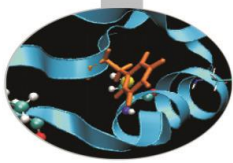
Esempio



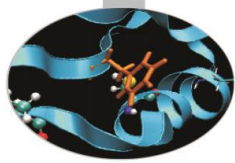
- output:

```
Costruttore Base: 4 5
Costruttore Base: 4 5
Costruttore Derivata: 6
Size of b: 8
Size of d: 12
Num1: 4
Num2: 5
Num3: 6
Sum: 15
Distruttore Derivata: 6
Distruttore Base: 4 5
Distruttore Base: 4 5
```

Ereditarietà e composizione



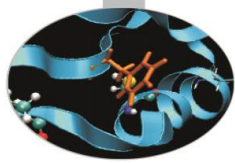
- Composizione ed ereditarietà rappresentano le due modalità per costruire una nuova classe ponendo al suo interno attributi e metodi di un'altra classe. Le istanze della nuova classe vengono a contenere un *sotto-oggetto*.
- Entrambe utilizzano una procedura di *inizializzazione del costruttore* per assegnare valori agli attributi del sotto-oggetto.
- L'ereditarietà appare più versatile. Si ricorre alla composizione qualora si sia principalmente interessati agli attributi del sotto-oggetto piuttosto che alla sua interfaccia pubblica. Dichiarando **private** il sotto-oggetto, l'utente della nuova classe viene forzato a non usare i metodi *public* del sotto-oggetto stesso.
- L'ereditarietà definisce una relazione **is-a** tra due classi; la composizione, invece, una relazione **has-a**.



Esempio

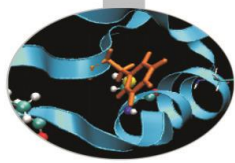
- Esempio: composizione

```
// file base.h
class Base{
  private:
    int num1;
    int num2;
  public:
    Base(int i1, int i2){num1=i1; num2=i2;}
    void setNum1(int i){ num1=i; }
    void setNum2(int i){ num2=i; }
    int getNum1(){ return num1; }
    int getNum2(){ return num2; }
};
```



Esempio

```
// file derivata.h
#include "base.h"
class Derivata{
    private:
      Base b;
      int num3;
    public:
      Derivata(int i1, int i2, int i3):b(i1,i2){num3=i3;}
      void setNum3(int i3){num3=i3;}
      int getNum3(){return num3;}
      int somma(){
        int sum;
        sum=num3+b.getNum2()+b.getNum1();
        return sum;
      }
};
```



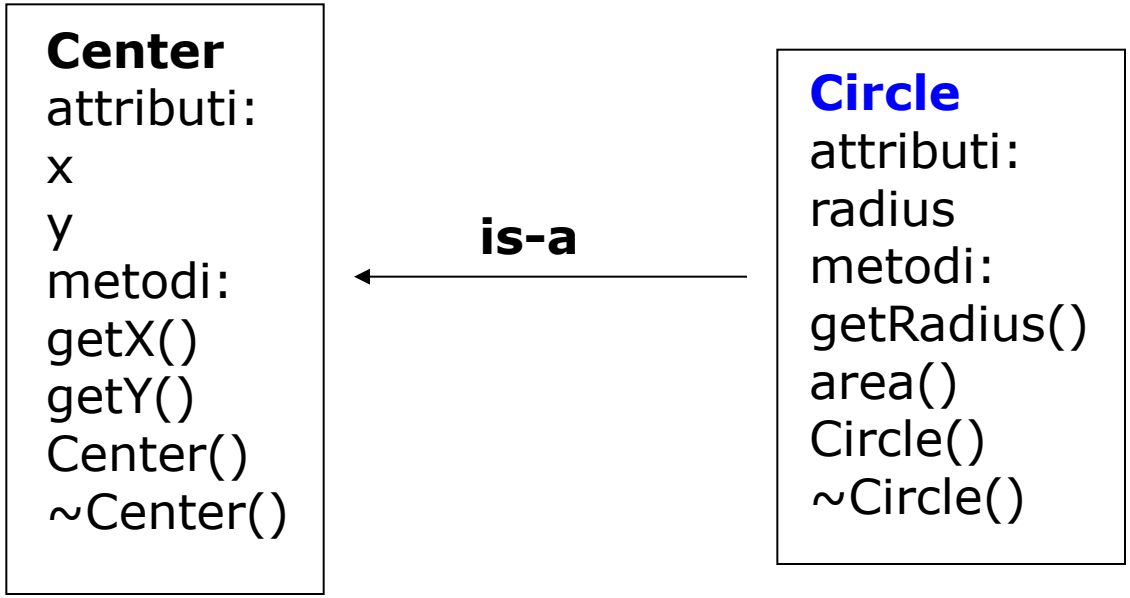
Esempio

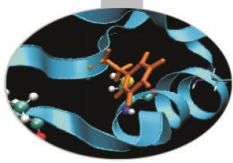
```
// file main.cpp
#include<iostream>
#include "derivata.h"
using namespace std;

int main(){
    Derivata d(4,5,6);
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "d->Num3: " << d.getNum3() << endl;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

- **Output:**

```
Size of d: 12
d->Num3: 6
Sum: 15
```

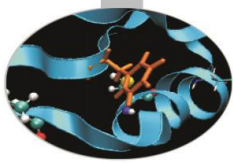
Un altro esempio di eredità pubblica





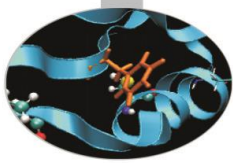
Esempio: center.h

```
#ifndef CENTER_H
#define CENTER_H
#include<iostream>
using namespace std;
class Center{
   friend ostream& operator<<( ostream&, const Center& );
   private:
     int x,y;
   public:
     Center(int=0, int=0);
     ~Center(){};
     int getX() const {return x;}
     int getY() const {return y;}
};
Center::Center(int x_pt, int y_pt){ x = x_pt;  y = y_pt; }
ostream& operator<<(ostream& output, const Center& ctr){
        output << "The center is: " << "("  << ctr.x << ","
       << ctr.y << ")" << endl;
        return output;
} #endif
```



Esempio: circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "center.h"
#include "math.h"
class Circle : public Center{
      friend ostream& operator<<(ostream&, const Circle&);
      private:
          double radius;
      public:
          Circle(double=0.0, int=0, int=0);
          ~Circle(){};
          double area() const {return 3.141593*pow(radius,2);}
};
Circle::Circle(double rad, int x_p, int y_p): Center(x_p, y_p){
          radius = rad;
}
ostream& operator<<(ostream& output, const Circle& cir){
   output  << static_cast<Center>(cir) ;
   output  << "The circle has radius = " << cir.radius
           << " and area= " << cir.area() << endl;
   return output; }#endif
```



Esempio

//file main.cpp

```
#include "circle.h"
int main(){
        Center centro(2,7);
        Circle cerchio(3.2, centro.getX(), centro.getY());
        cout << cerchio;
        return 0;
}
```

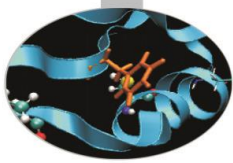
Output

```
The center is: (2,7)
The circle has radius = 3.2 and area= 32.1699
```

- Sfruttando il fatto che un oggetto della classe derivata Circle è un (*is-a*) oggetto della classe base Center abbiamo potuto usare, all'interno del file circle.h, l'istruzione:

  - `output << static_cast<Center>(cir);`

che richiama automaticamente l'overloading dell'operatore << per la classe Center e di conseguenza permette di stampare su standard output la sola parte dell'oggetto cir ereditata dalla sua classe base (il sotto-oggetto), ovvero il suo centro.

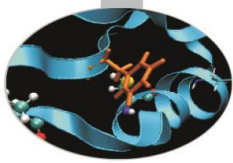


L'eredità di tipo private

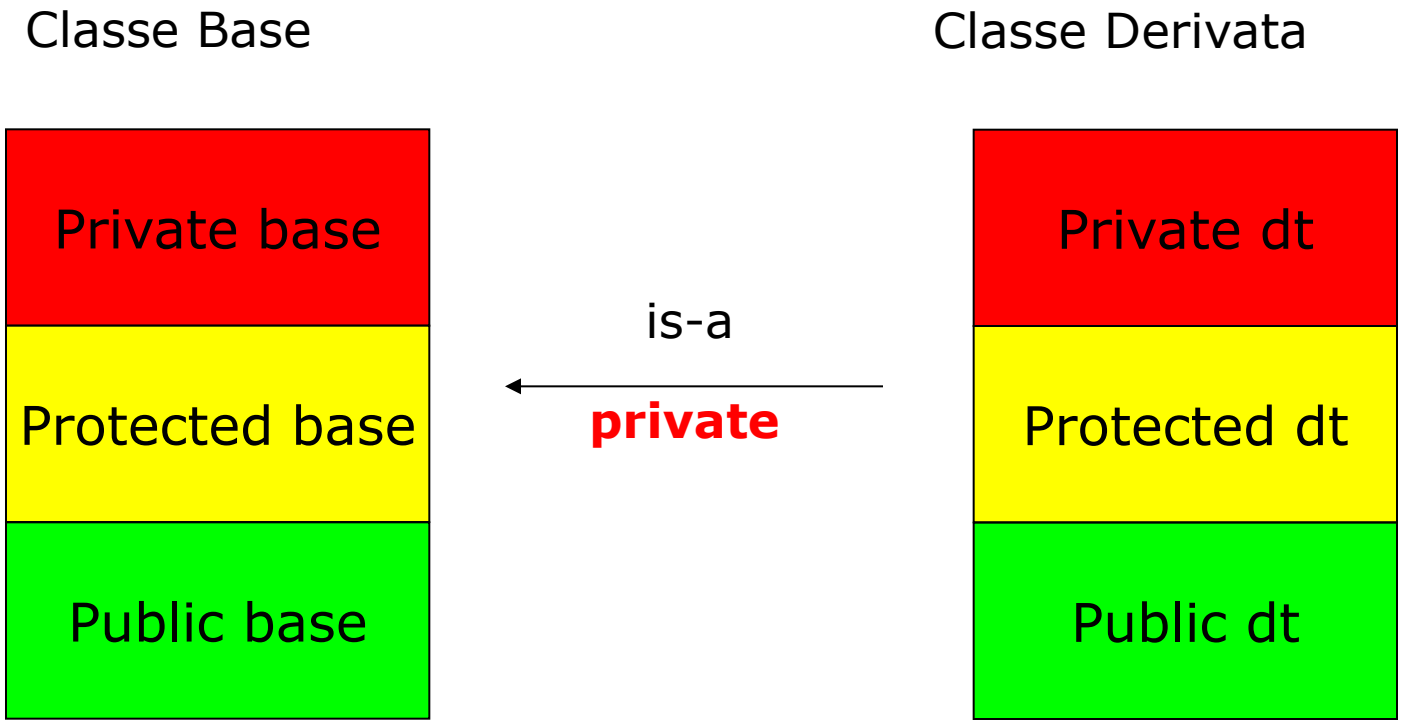
- La dichiarazione di ereditarietà di tipo private è del tutto analoga a quella di tipo public:

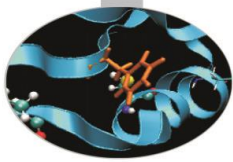
```
nome_classe_derivata : private nome_classe_base{
     corpo della classe derivata;
};
```

- Nell'eredità di tipo private, la classe derivata vede sia i membri ***public*** che quelli ***protected*** della classe base come ***private***: essi sono accessibili direttamente **solo** dall'interno della classe derivata.
- I membri ***private*** della classe base ***non*** sono invece, direttamente accessibili dalla classe derivata.
- L'ereditarietà di tipo private è un procedimento del tutto analogo alla composizione perché vieta all'utente di utilizzare i metodi della classe base tramite gli oggetti della classe derivata dal momento che quest'ultima li tratta come private.

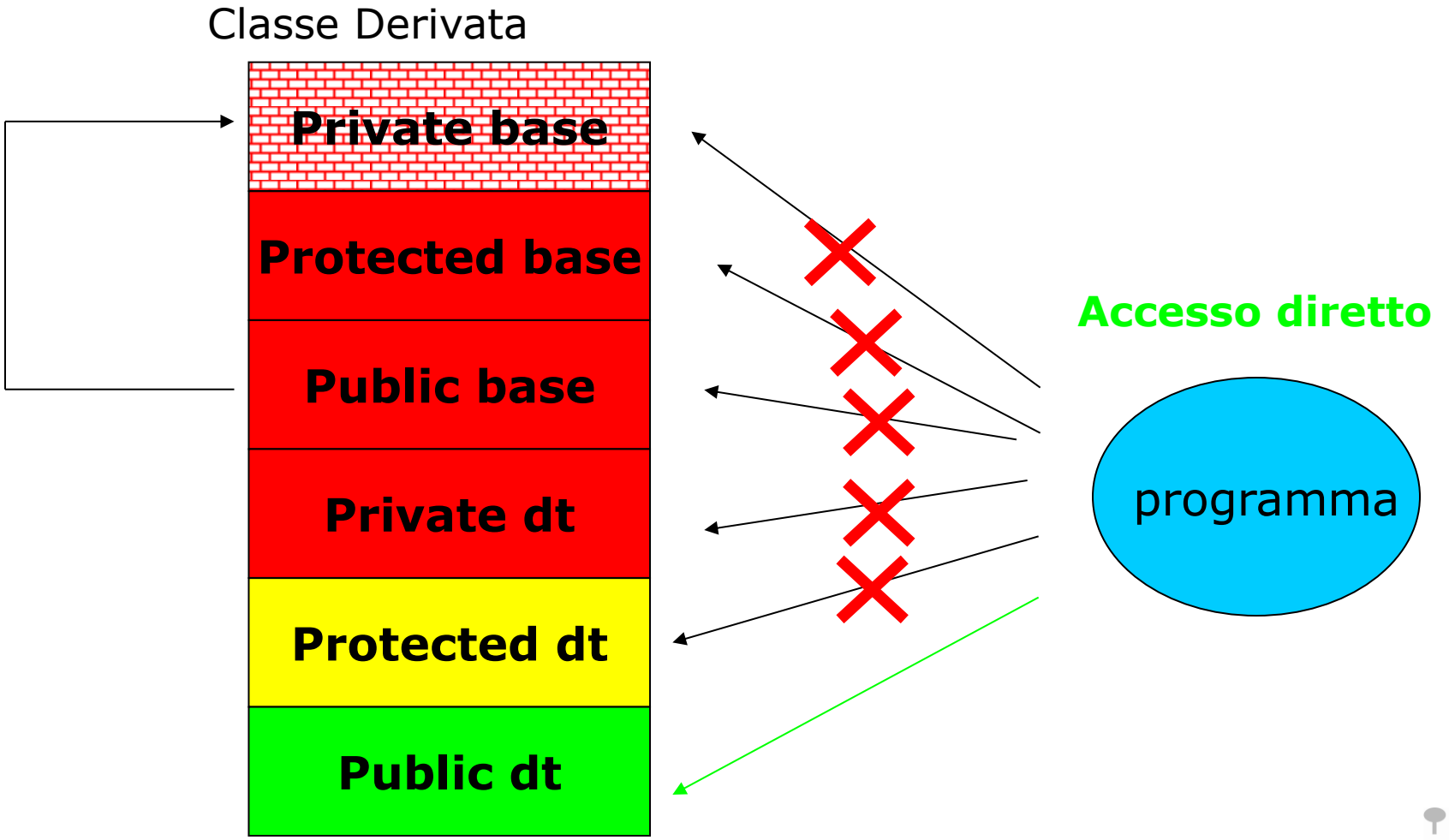


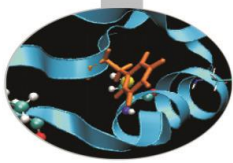
L'eredità di tipo private



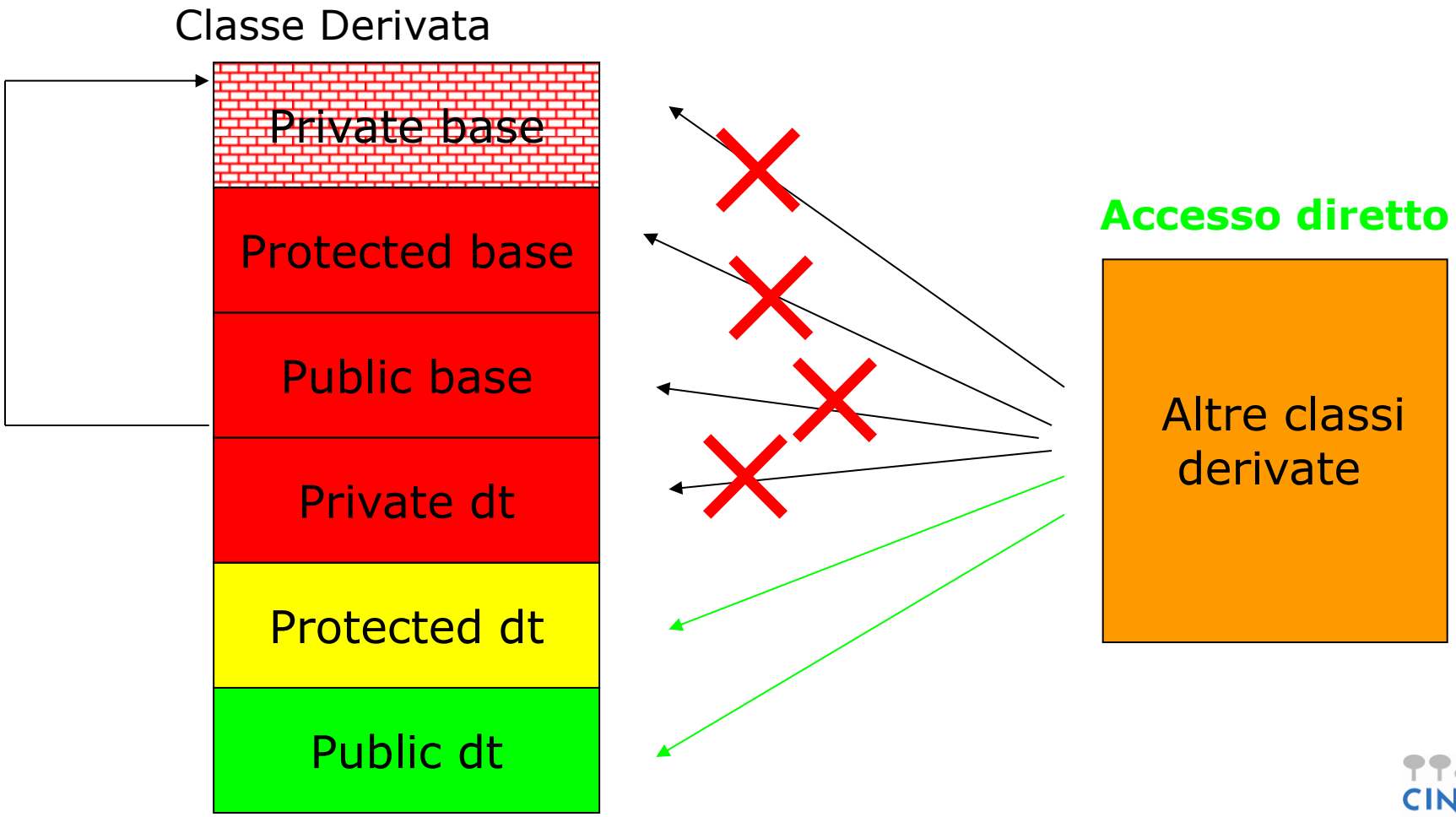


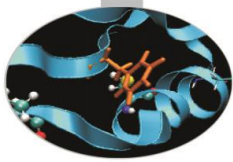
L'eredità di tipo private





L'eredità di tipo private

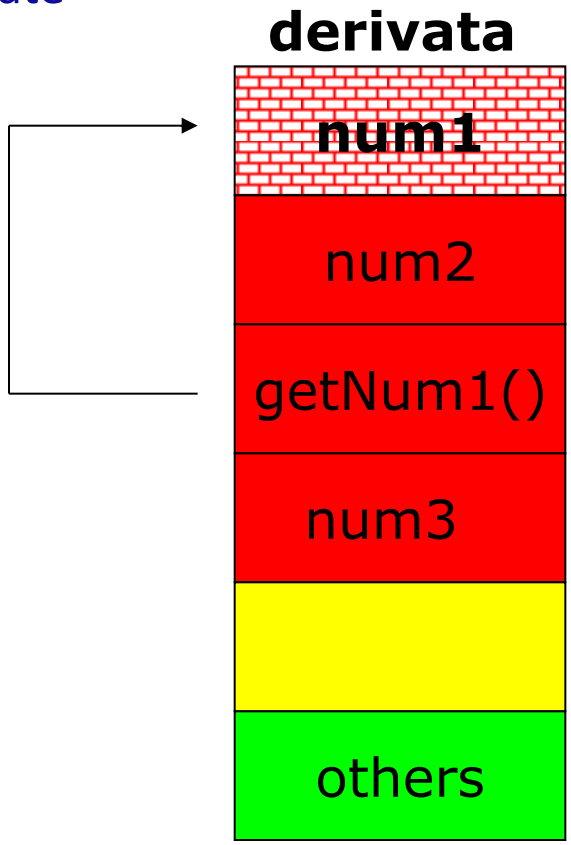


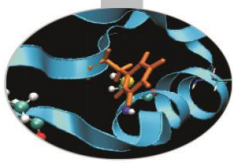


Esempio

- esempio: sintassi nell'ereditarietà di tipo private

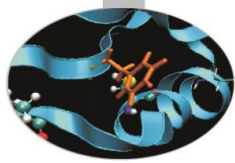
```
// file base.h
class Base{
   private:
      int num1;
   protected:
      int num2;
   public:
      Base(){num1=0; num2=0;}
      void setNum1(int i){ num1=i; }
      void setNum2(int i){ num2=i; }
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
};

};
```





Esempio

```
// file derivata.h
#include "base.h"
#include<iostream>
using namespace std;
class Derivata: private Base{
    private:
      int num3;
    public:
      Derivata(int i1, int i2){setNum1(i1); setNum2(i2); num3=0;
   }
      void setNum3(int i){num3=i;}
      int getNum3(){return num3;}
      void printBaseComp(){
        cout << "d->Num1: " << getNum1() << " Num2: " << num2 <<
   endl;}
      int somma(){int sum; sum=num3+num2+getNum1(); return sum;}
```

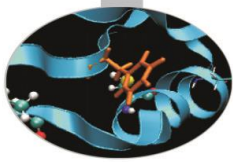


Esempio

```
// file main.cpp
#include "derivata.h"
int main(){
    Derivata d(4,5);
    d.setNum3(6);
    cout << "Size of d: " << sizeof(d) << endl;
    d.printBaseComp();
    cout << "d->Num3: " << d.getNum3() << endl;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

Output:

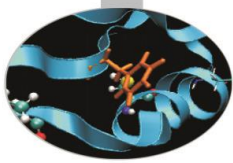
```
Size of d: 12
d->Num1: 4 Num2: 5
d->Num3: 6
Sum: 15
```



Esempio

- Per rendere visibili all'utente attributi o metodi della classe base è sufficiente riscrivere il loro nome (preceduto da quello della classe base e dall'operatore ::) nell'interfaccia pubblica della classe derivata.

```
// file derivata.h
// la funzione printBaseComp() non serve più
#include "base.h"
#include<iostream>
using namespace std;
class Derivata: private Base{
    private:
      int num3;
    public:
      Derivata(int i1, int i2){ setNum1(i1); setNum2(i2); num3=0;
   }
      void setNum3(int i){num3=i;}
      int getNum3(){return num3;}
      Base::getNum1;
      Base::getNum2;
      int somma(){
        int sum;
        sum=num3+num2+getNum1();
        return sum; }};
```

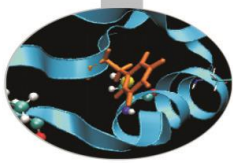


```
// file main.cpp
#include "derivata.h"
int main(){
    Derivata d(4,5);
    d.setNum3(6);
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "d-Num1: " << d.getNum1() << " d->Num2: "
         << d.getNum2()
         << " d->Num3: " << d.getNum3() << endl;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

Output:

```
Size of d: 12
d-Num1: 4 d->Num2: 5 d->Num3: 6
Sum: 15
```

L'ereditarietà di tipo protected

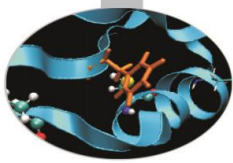


- Dichiarazione di ereditarietà di tipo protected:

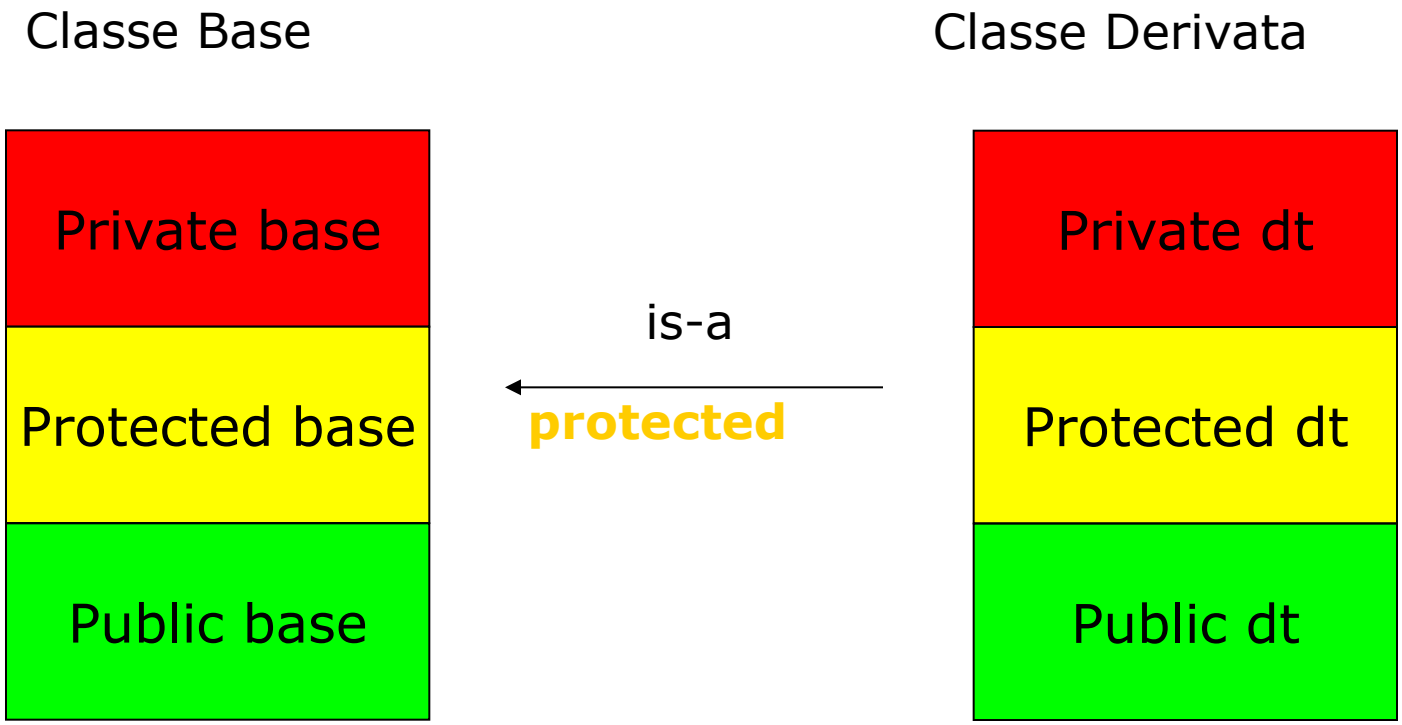
```
nome_classe_derivata : protected nome_classe_base{
     corpo della classe derivata;
};
```

- Nell'eredità di tipo protected, la classe derivata vede sia i membri ***public*** che quelli ***protected*** della classe base come ***protected***: essi **non** sono accessibili direttamente dall'esterno, ma unicamente dall'interno di ogni classe derivata.

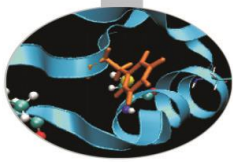
- L'ereditarietà di tipo protected non è molto usata: esiste solo per completezza di linguaggio.

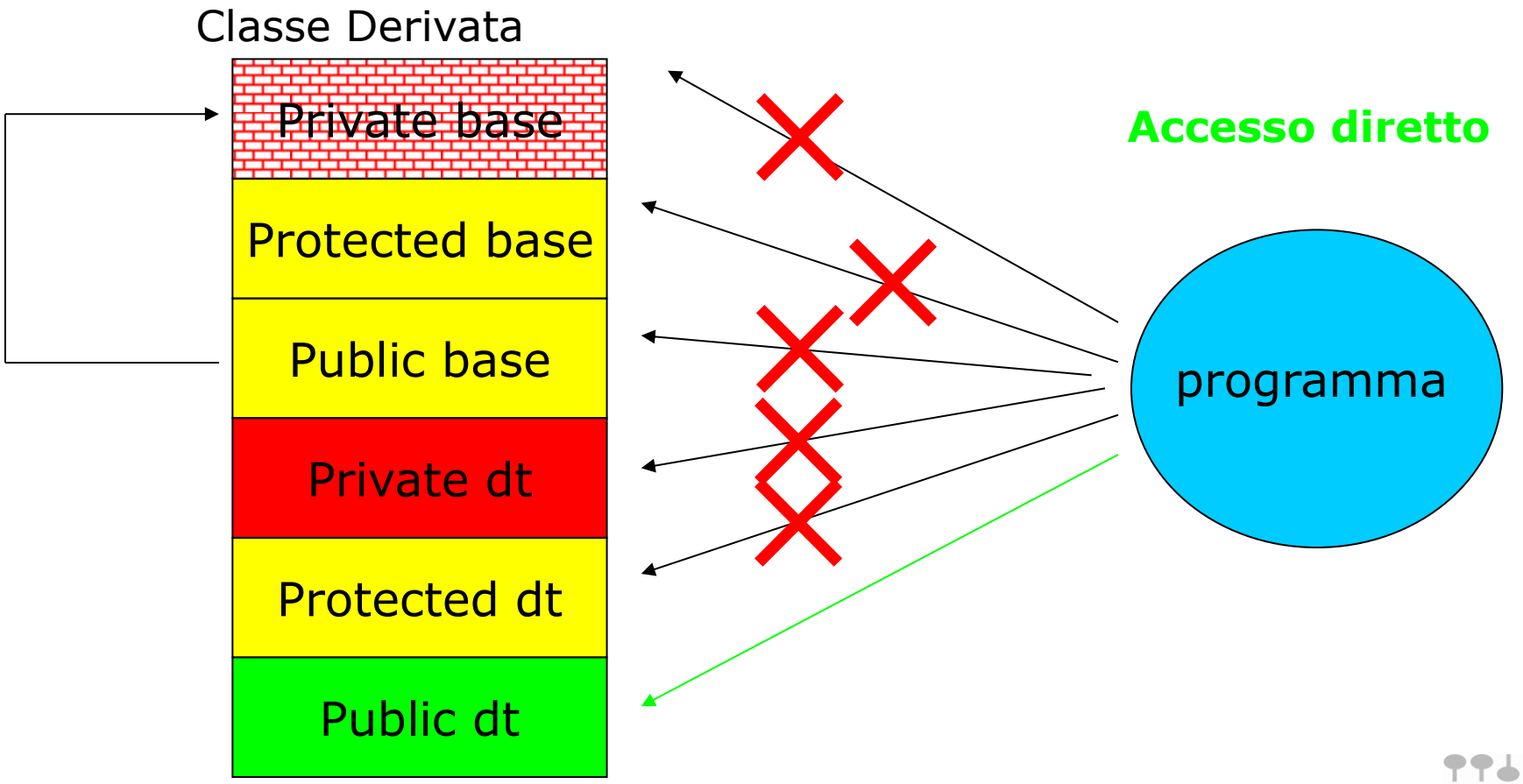


L'ereditarietà di tipo protected

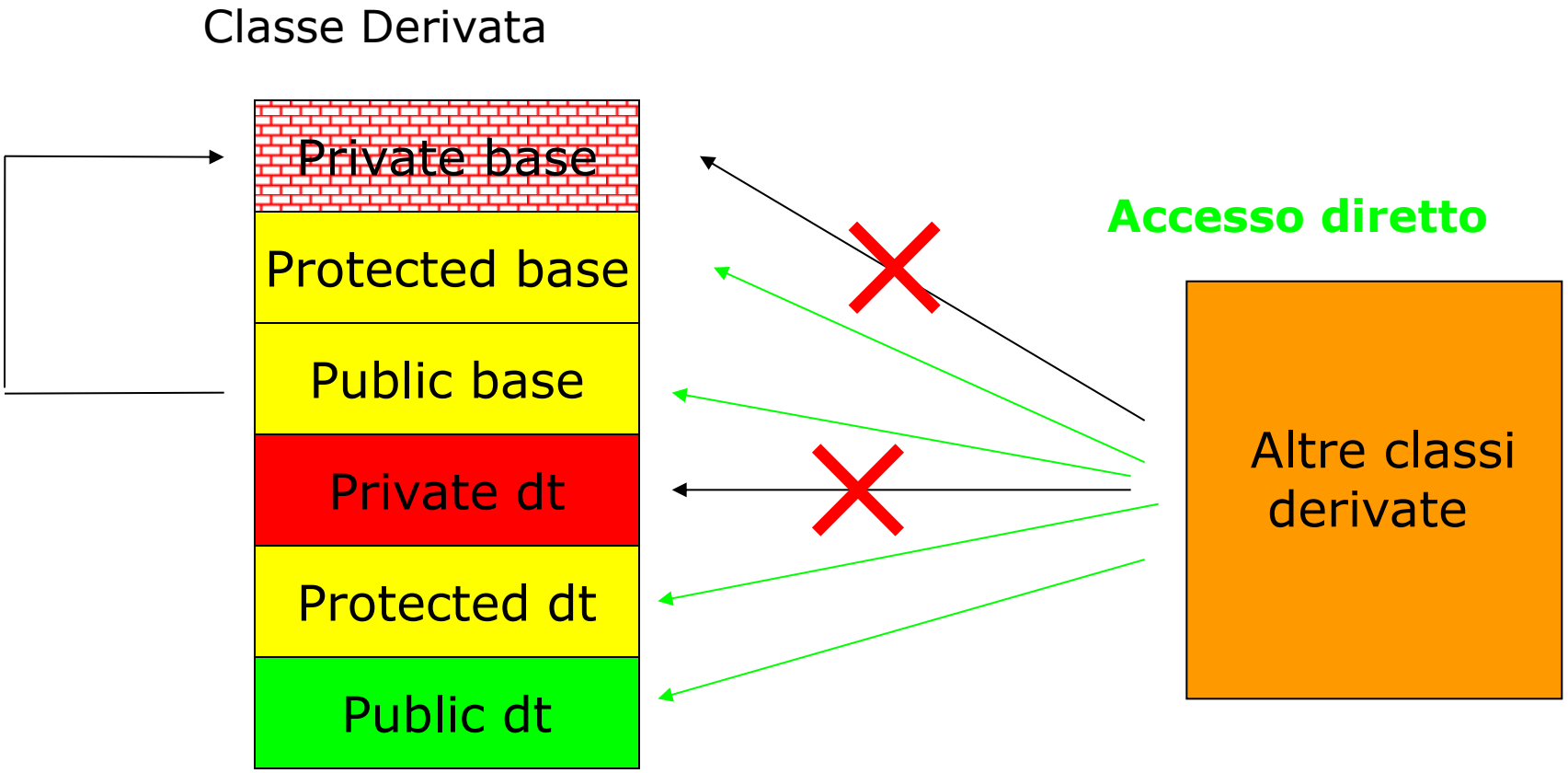


L'ereditarietà di tipo protected

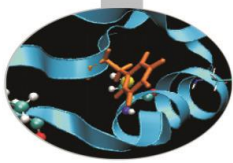




L'ereditarietà di tipo protected

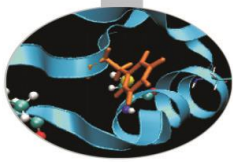


I tre tipo di ereditarietà a confronto



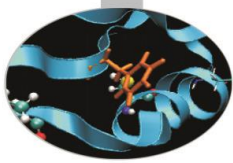
Tipo di ereditarietà	Classe Base	Classe Derivata
Public	public	public
	protected	protected
	private	***hidden***
Protected	public	***protected***
	protected	protected
	private	***hidden***
Private	public	***private***
	protected	***private***
	private	***hidden***

L'overriding dei metodi della classe base nelle classi derivate



- Una classe derivata può eseguire l'***overriding***, ovvero la sovrascrittura, di un metodo della classe base implementandone una nuova versione con uguali: tipo restituito, nome, attributi e lista di argomenti.
- L'overriding è una procedura simile all'overloading. Quest'ultimo, però, richiede che due riscritture della stessa funzione siano differenti solo per il numero o il tipo degli argomenti passati.
- Se viene eseguito l'overriding di una funzione overloaded nella classe base, la classe derivata avrà accesso diretto solo alla versione di cui ha fatto l'overriding.
- Qualora si rendesse, tuttavia, necessario chiamare un metodo della classe base nascosto alla classe derivata a causa del suo overriding, basterà richiamare la funzione facendo precedere il suo nome dall'operatore di risoluzione dello scope **::** nonché dal nome della classe base:

```
nome_classe_base :: nome_funzione_membro( argomenti )
```

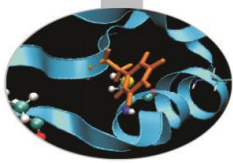


Esempio: base.h

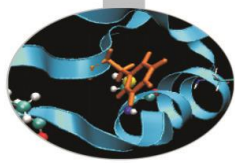
- **Esempio: overriding e overloading**

```
// file base.h
#ifndef BASE_H
#define BASE_H
#include<iostream>
using namespace std;
class Base{
   private:
      int num1;
      int num2;
   public:
      Base(int i1, int i2){ num1=i1; num2=i2; }
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
```

Esempio: base.h

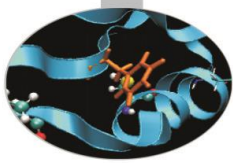


```
void print(){
        cout << "Base::print()" << endl;
        cout << "Num1: " << num1 << " Num2: " << num2 << endl;
      }
      void print(int i){
        cout << "Base::print(int)" << endl;
        cout << "Int: " << i << endl;
        cout << "Num1 and Num2: " << num1 << " " << num2 << endl;
      }
};
#endif
```



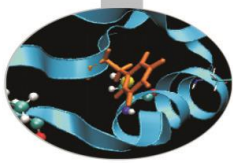
Esempio: derivata1.h

```
#include "base.h"
class Derivata1: public Base{
    private:
      int num3;
      int sum;
    public:
      Derivata1(int i1, int i2, int i3):Base(i1,i2){num3=i3;
   sum=0;}
      int somma(){
        sum=num3+getNum2()+getNum1();
        return sum;
      }
      void print(){
        cout << "Derivata1::print()" << endl;
        cout << "Num1: " << getNum1() << " Num2: "<< getNum2()
             << " Num3 : " << num3 << endl;
        cout << "Sum: " << sum << endl;
      }};
```



Esempio: derivata2.h

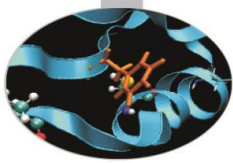
```
#include "base.h"
class Derivata2: public Base{
    private:
      int num3;
      int sum;
    public:
      Derivata2(int i1, int i2, int i3):Base(i1,i2){num3=i3;
   sum=0;}
      int somma(){ sum=num3+getNum2()+getNum1(); return sum; }
      void print(){
        cout << "Derivata2::print()" << endl;
        cout << "Sum is:" << sum <<endl; }
      void print(char dummy){
        cout << "Derivata2::print(char)" << endl;
        cout << "Char: " << dummy << endl;
        cout << "Num1: " << getNum1() << " Num2: " << getNum2()
             << " Num3 : " << num3 << endl;
        cout << "Sum: " << sum << endl;
      } };
```



Esempio: main.ccp

```
#include "derivata1.h"
#include "derivata2.h"
int main(){
  Base b(4,5);
  Derivata1 d1(b.getNum1(),b.getNum2(),6);
  Derivata2 d2(b.getNum1(),b.getNum2(),10);
  d1.somma();
  d2.somma();
  cout << "d1 printing:" << endl;
  d1.print();
  d1.Base::print();
  d1.Base::print(d1.somma());
  cout << endl;
  cout << "d2printing:" << endl;
  d2.print();
  d2.print('c');
  d2.Base::print();
  d2.Base::print(d2.somma());
  return 0;}
```

Esempio: output

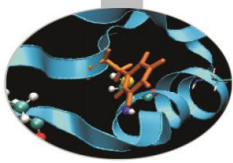


- **output:**

```
d1 printing:
Derivata1::print()
Num1: 4 Num2: 5 Num3 : 6
Sum: 15
Base::print()
Num1: 4 Num2: 5
Base::print(int)
Int: 15
Num1 and Num2: 4 5
d2printing:
Derivata2::print()
Sum is:19
Derivata2::print(char)
Char: c
Num1: 4 Num2: 5 Num3 : 10
Sum: 19
```

```
Base::print()
Num1 4 Num2: 5
Base::print(int)
Int: 19
Num1 and Num2: 4 5
```

Ereditarietà multipla

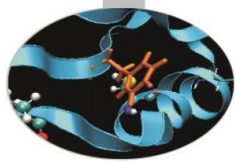


- Una classe derivata può avere più di una classe base: in questo caso si parla di *ereditarietà multipla*

```
class classe_derivata : public classe_base1, … , public
  classe_baseN
```

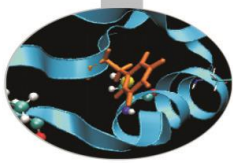
- E' possibile che due o più classi base contengano funzioni membro dichiarate con lo stesso nome. Per chiamare queste funzioni senza alcuna ambiguità bisogna ricorrere all'operatore di risoluzione dello scope **::** preceduto dal nome della corrispondente classe base:

```
nome_oggetto.classe_baseA :: nome_funzione;
nome_oggetto.classe_baseB :: nome_funzione;
```



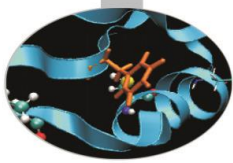
Ereditarietà multipla

```
// file base1.h
class Base1{
   private:
      int num1;
      int num2;
   public:
      Base1(int i1, int i2){num1=i1; num2=i2;}
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
};
// file base2.h
class Base2{
   private:
      int num1;
      int num2;
   public:
      Base2(int i1, int i2){num1=i1; num2=i2;}
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
};
```



Ereditarietà multipla

```
// file derivata.h
#include "base1.h"
#include "base2.h"

class Derivata: public Base1, public Base2{
    private:
      int num3;
    public:
      Derivata(int i1, int i2, int i3, int i4, int i5):
        Base1(i1,i2),Base2(i3,i4){ num3=i5; }
      int getNum3(){ return num3;}
      int somma(){
        int sum;

   sum=num3+Base2::getNum2()+Base2::getNum1()+Base1::getNum2()+
                  Base1::getNum1();
        return sum;
      }
};
```

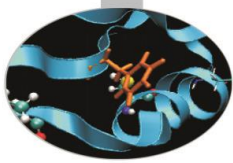


Ereditarietà multipla

```
// file main.cpp
#include<iostream>
#include "derivata.h"
using namespace std;

int main(){
    Derivata d(4,5,6,7,8);
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "d->Num1: " << d.Base1::getNum1() << " Num2: "
         << d.Base1::getNum2() << " Num3: " << d.Base2::getNum1()
         << " Num4: " << d.Base2::getNum2() << " Num5: " <<
   d.getNum3()
         <<endl;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

Output:

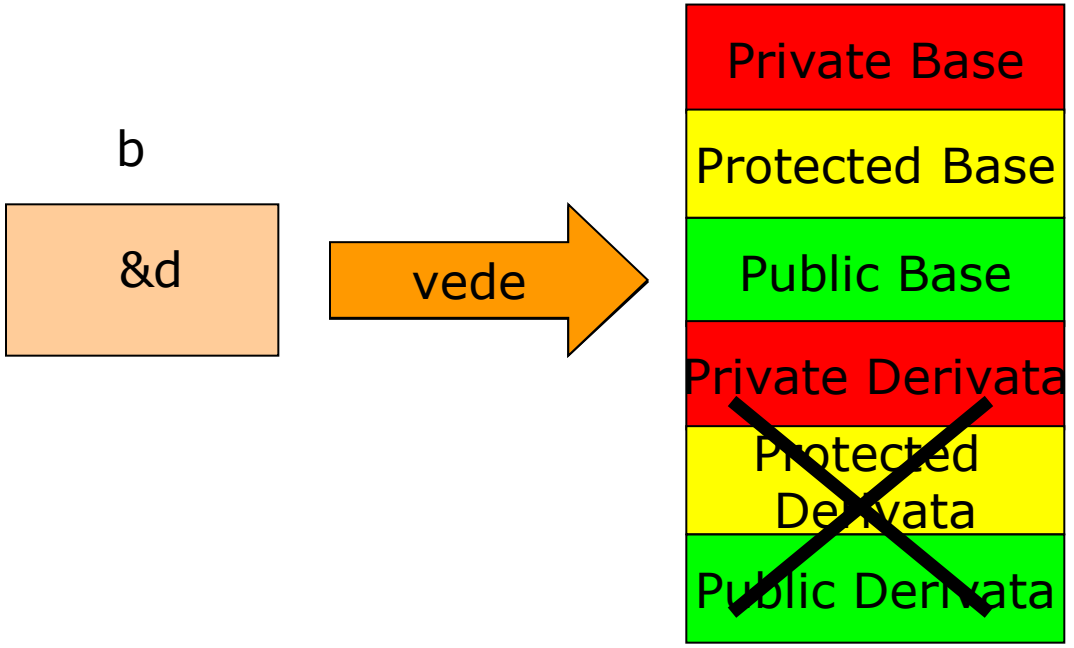
```
Size of d: 20
d->Num1: 4 Num2: 5 Num3: 6 Num4: 7 Num5: 8
Sum: 30
```



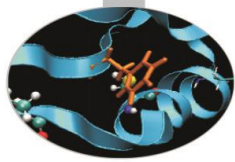
Casting

- Nell'ereditarietà (di tipo public) un oggetto della classe derivata non è altro che una particolare istanza della classe base.
- E' permesso assegnare ad un puntatore alla classe base l'indirizzo di un oggetto della classe derivata. Nell'operazione di assegnamento, l'oggetto puntato subisce una conversione di tipo (derivata -> base). Il puntatore alla classe base è in grado di vedere solo i membri ed i metodi ereditati dalla classe base stessa da parte dell'oggetto puntato.

```
Base* b;
Derivata d;
b = &d;
```

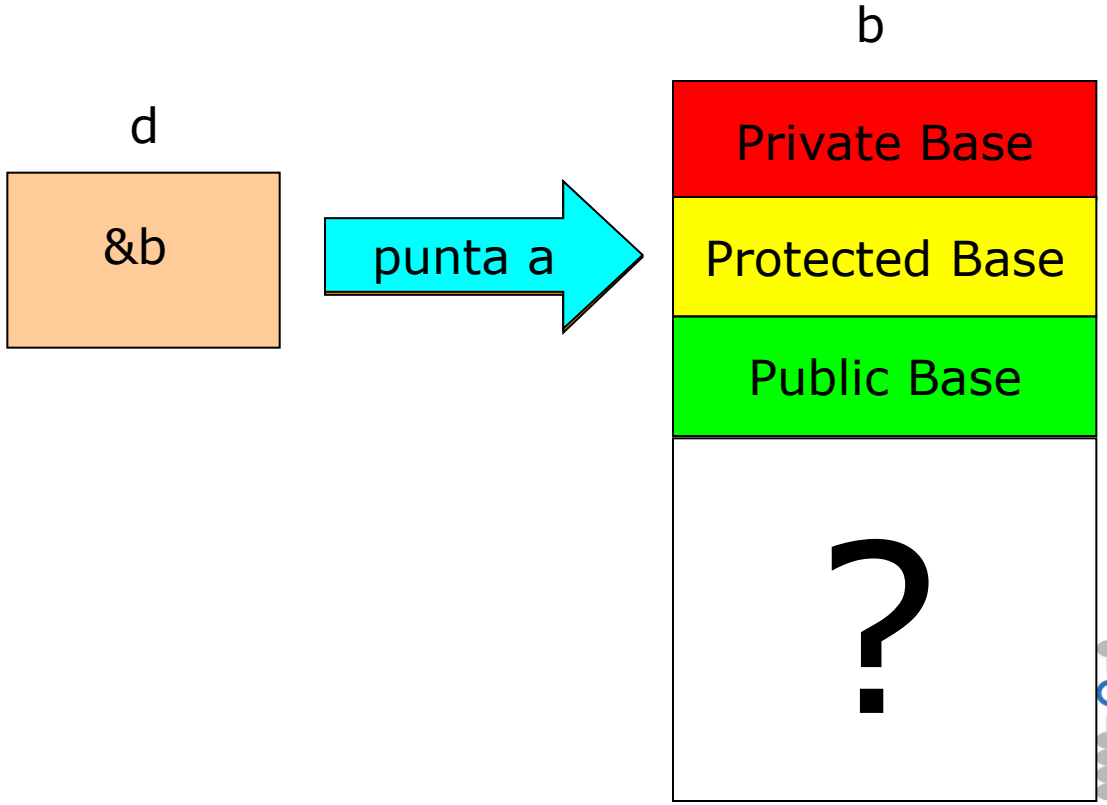


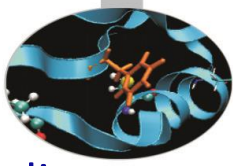
Casting



- E', al contrario, vietato assegnare direttamente ad un puntatore alla classe derivata l'indirizzo di un oggetto della classe base. Il casting automatico (base -> derivata) creerebbe, infatti, un oggetto incompleto.

```
Base b;
Derivata* d;
d=&b;
```





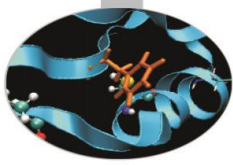
Casting

- E', tuttavia, possibile forzare il compilatore ad eseguire un'operazione di casting (base -> derivata) facendo uso dell'operatore **`static_cast`**:

```
Base b;
Derivata* d;
d=static_cast<Derivata*>(&b);
```

- L'operatore **`dynamic_cast`**, invece, non permette di svolgere alcuna conversione di tipo che porti alla creazione di oggetti incompleti. Tale operatore non accetta il casting (base -> derivata) a meno che la classe base non sia polimorfa.

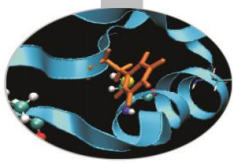
```
Base b;
Derivata* d;
d=dynamic_cast<Derivata*>(&b); // errore (base->derivata)
Derivata d;
Base* b;
b=dynamic_cast<Base*>(&d); // ok (derivata->base)
```



Esempio: base.h

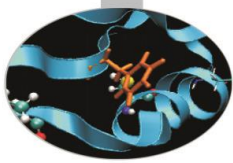
- Esempio1: casting automatico (derivata->base)

```
//file base.h
#include<iostream>
using namespace std;

class Base{
   friend ostream& operator<<(ostream&, const Base&);
   private:
      int num1;
      int num2;
   public:
      Base(int i1, int i2){num1=i1; num2=i2;}
      int getNum1(){ return num1; }
      int getNum2(){ return num2; }
};
ostream& operator<< (ostream& out, const Base& b){
        out << "Num1-> " << b.num1 << " Num2-> "
            << b.num2 << endl;
        return out;}
```



Esempio: derivata.h

```
#include "base.h"
class Derivata: public Base{
    friend ostream& operator<<(ostream&, const Derivata&);
    private:
      int num3;
    public:
      Derivata(int i1, int i2, int i3):Base(i1,i2){num3=i3;}
      int getNum3(){ return num3;}
      int somma(){
        int sum;
        sum=num3+getNum2()+getNum1();
        return sum;
      }
};
ostream& operator<<(ostream& out, const Derivata& d){
        out << static_cast<Base>(d);
        out << "Num3-> " << d.num3 << endl;
        return out;}
```



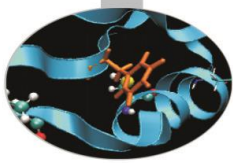
Esempio: main.ccp

```
#include "derivata.h"
int main(){
    Base* b;
    Derivata d(4,5,6);
    b=&d;
    cout << "Size of b: " << sizeof(b) << endl;
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "Base: " << *b << endl;
    cout << "Derivata: " << d;
    cout << "Sum: " << d.somma() << endl;
    return 0;
}
```

- output:

```
Size of b: 4
Size of d: 12
Base: Num1-> 4 Num2-> 5

Derivata: Num1-> 4 Num2-> 5
Num3-> 6
Sum: 15
```

(è un puntatore)



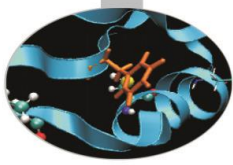
Esempio: derivata.h

- **Esempio2: casting errato (base->derivata)**

```
// file main.cpp
#include "derivata.h"
int main(){
    Base b(4,5);
    Derivata* d;
    d=&b;
    cout << "Size of b: " << sizeof(b) << endl;
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "Base: " << b << endl;
    cout << "Derivata: " << *d;
    cout << "Sum: " << d->somma() << endl;
    return 0;
}
```

- compiling:

```
main.cpp: In function int main() :
main.cpp:6: error: invalid conversion from Base* to Derivata*
```



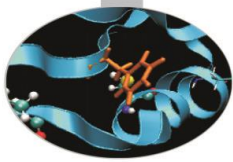
Esempio: main.ccp

- **Esempio3: dynamic_cast errato (base->derivata)**

```
// file main.cpp
#include "derivata.h"
int main(){
    Base b(4,5);
    Derivata* d;
    d=dynamic_cast<Derivata*>(&b);
    cout << "Size of b: " << sizeof(b) << endl;
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "Base: " << b << endl;
    cout << "Derivata: " << *d;
    cout << "Sum: " << d->somma() << endl;
    return 0;
}
```

- compiling:

```
main.cpp: In function int main() :
main.cpp:6: error: cannot dynamic_cast &b (of type class Base*)
   to type class Derivata* (source type is not polymorphic)
```



Esempio

- Esempio4: static_cast, oggetto incompleto (base->derivata)

```
// file main.cpp
#include "derivata.h"
int main(){
    Base b(4,5);
    Derivata* d;
    d=static_cast<Derivata*>(&b);
    cout << "Size of b: " << sizeof(b) << endl;
    cout << "Size of d: " << sizeof(d) << endl;
    cout << "Base: " << b << endl;
    cout << "Derivata: " << *d;
    cout << "Sum: " << d->somma() << endl;
    return 0;
}
```

- output:

```
Size of b: 8
Size of d: 4
Base: Num1-> 4 Num2-> 5
Derivata: Num1-> 4 Num2-> 5
Num3-> -1073748344
Sum: -1073748335
```

(è un puntatore)