# Ereditarietà e polimorfismo

Programmazione di Sistema

A.A. 2017-18





# Argomenti

- Ereditarietà
- Polimorfismo
- Conversione tra tipi



# Ereditarietà

- A volte, classi diverse presentano comportamenti simili
  - In quanto modellano concetti semanticamente simili
- Sebbene si possa duplicare il codice condiviso, questa raramente è una buona idea
  - Il codice diventa difficile da leggere e la manutenzione praticamente impossibile
- Una classe può essere definita come specializzazione di una classe esistente
  - Ereditandone variabili istanza e funzioni membro
- La classe così definita viene detta sotto-classe
  - Quella da cui deriva viene detta super-classe



# Ereditarietà

- La sotto-classe specializza il comportamento della super-classe
  - Aggiungendo ulteriori variabili istanza e funzioni membro

```
class File {
  int fileDescriptor;

public:
  uint_8 read ();

  size_t readBlock(
    uint_8 *ptr,
    size_t  offset,
    size_t  count);

  int close();
};
```

```
class TextFile: public
  File  {

  CharCodec  codec;

public:
  wchar_t readChar();

  size_t
  readCharBlock(
    wchar_t *ptr,
    size_t  offset,
    size_t  count);
};
```

# Ereditarietà multipla

- In C++, una classe può  ereditare da una o più classi
  - Specializzando così il concetto che esse rappresentano
  - Non esiste una classe antenata radice dalla quale tutte le classi sono derivate
- L'ereditarietà può essere
  - Pubblica
  - Privata
  - Protetta
- Se occorre, i metodi della classe di base possono essere invocati usando l'operatore "::" preceduto dal nome della classe base



# Ereditarietà

```
class Base {
public:
      Base();
      void baz();
     ~Base() {}
};
class Der : public Base {
public:
     Der():Base(){
     Base::baz()
   }
  ~Der() {}
};

//...
{
     Der *obj = new Der();
     obj->baz();
     delete obj;
}
```

**Attenzione!** In C++ non esiste la parola chiave “super”, non funzionerebbe con ereditarietà multipla

# Polimorfismo

- Se classe A estende in modo pubblico la classe B, è lecito assegnare ad una variabile “v” di tipo B* un puntatore ad un oggetto di tipo A
  - A <<is_a>> B
- Se la classe A ridefinisce il metodo “m()”, cosa succede quando si invoca v->m() ?
  - Dipende da come è stato dichiarato il metodo nella classe base



# Polimorfismo

```
class B {
  public:
    int m() { return 1; }
};

class A: public B {
  public:
    int m() { return 2; }
};

int main(int argc, char** argv) {
      B* ptr = new A();
      return ptr->m(); // ???
}
```



# Polimorfismo

- Per default, il C++ utilizza l'implementazione definita nella classe a cui ritiene appartenga l'oggetto
- Nel caso precedente, ptr ha come tipo B*
  - Per cui il compilatore invocherà la definizione di m() contenuta nella classe B
  - Anche se, di fatto, l'oggetto cui ptr punta è di classe A
- Per modificare questo comportamento, occorre anteporre alla definizione del metodo

  la parola chiave virtual
  - Abilitando così il funzionamento polimorfico



# Metodi virtuali

- Solo le funzioni denominate “virtual” sono polimorfiche
- Le funzioni virtuali astratte sono dichiarate
  “ = 0; ”
- Una classe astratta contiene almeno una funzione virtuale astratta
  - Una classe puramente astratta contiene solo funzioni virtuali astratte
  - Equivalente ad una interfaccia Java



# Metodi virtuali

```
class Base {
public:
  Base() {}
  virtual void foo() { … }
  virtual void bar() = 0;
  virtual ~Base() {}
};

class Der : public Base {
public:
  Der() : Base() { … }
  void foo(){
    Base::foo();
  }
  void bar() { … }
  ~Foo() {}
};
```

**Attenzione!**
In Java,
al contrario, i metodi non polimorfici vanno esplicitamente dichiarati **“final”**

# Distruttori virtuali

- Anche i distruttori non sono polimorfici per default
- È opportuno che tutte le classi con funzioni virtuali abbiano il distruttore dichiarato virtual
- Se una classe con metodi virtuali non ha un distruttore virtuale
  - è possibile che venga chiamato il distruttore errato

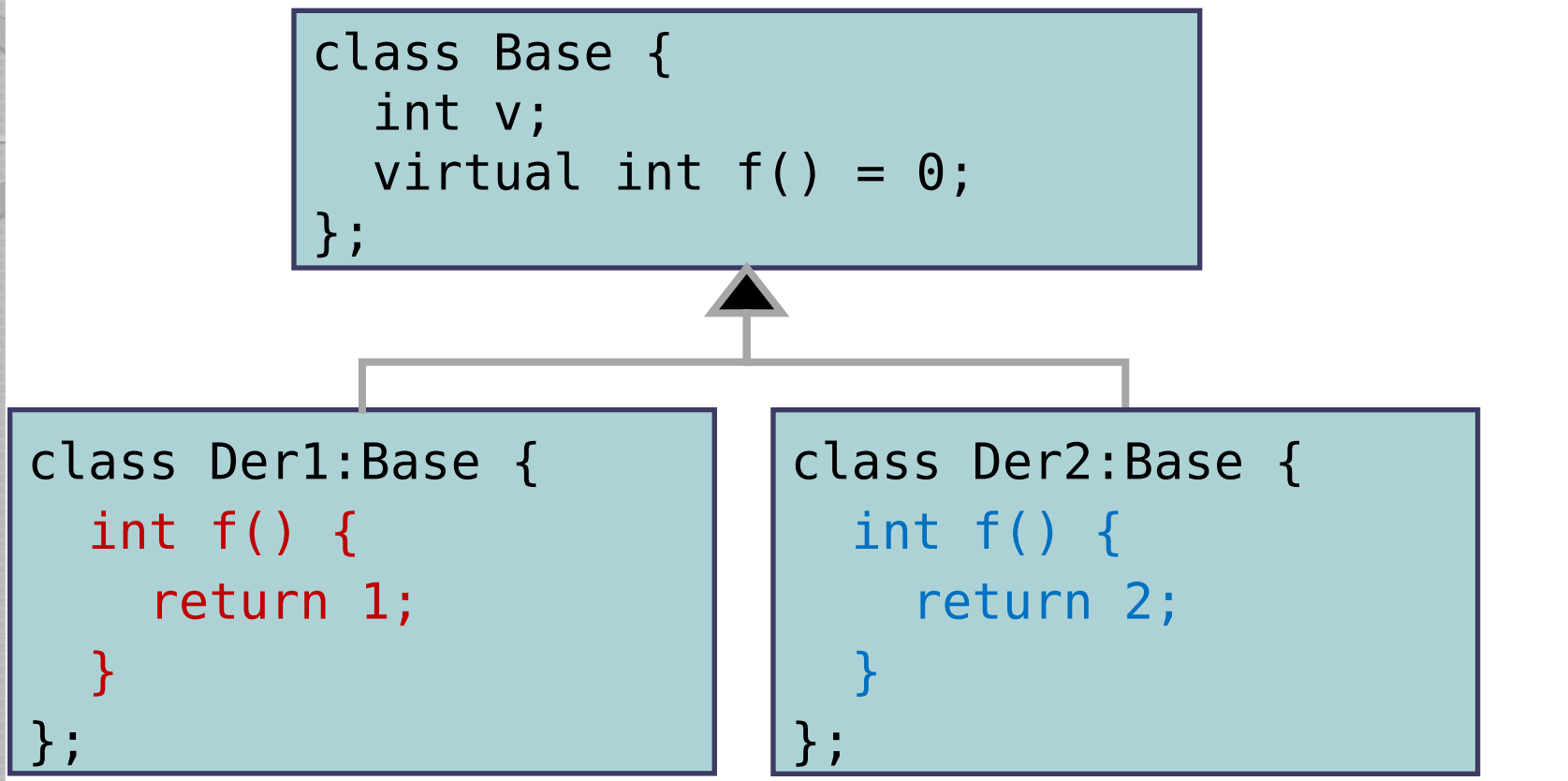


# Distruttori virtuali

```
class Base {
public:
  Base() {}
  virtual void foo() { … }
  virtual void bar() = 0;
  virtual ~Base() { … }
};

class Der : public Base {
public:
  Der() : Base() { … }
  void foo(){
    Base::foo();
  }
  void bar() { … }
  ~Der() { … }
};
```



# Polimorfismo




```
Base* b1= new Der1();
Base* b2= new Der2();

printf("%d\n", b1->f());
printf("%d\n", b2->f());
```

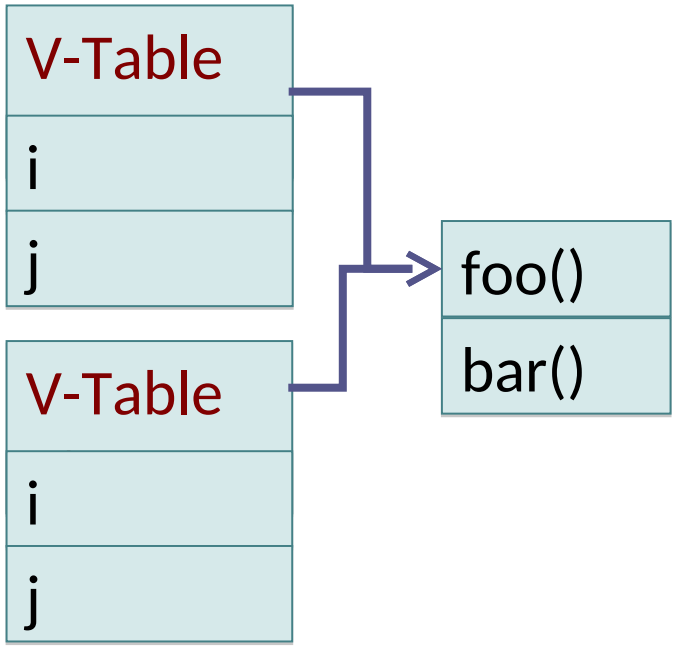
# Metodi virtuali e V-Table

- Dato che b1 e b2 hanno lo stesso tipo (Base*), come fa il compilatore a decidere quale implementazione del metodo f() invocare?
- Per i metodi virtuali,  si introduce un livello di indirezione attraverso un campo nascosto detto
  V-Table
- Ogni volta che viene creato un oggetto, nella memoria allocata si aggiunge un puntatore
  - Punta ad una tabella (statica) che contiene tante righe quanti sono i metodi virtuali dell'oggetto creato
  - Ciascuna riga viene riempita con il puntatore all'effettivo metodo virtuale



# Metodi virtuali e V-Table

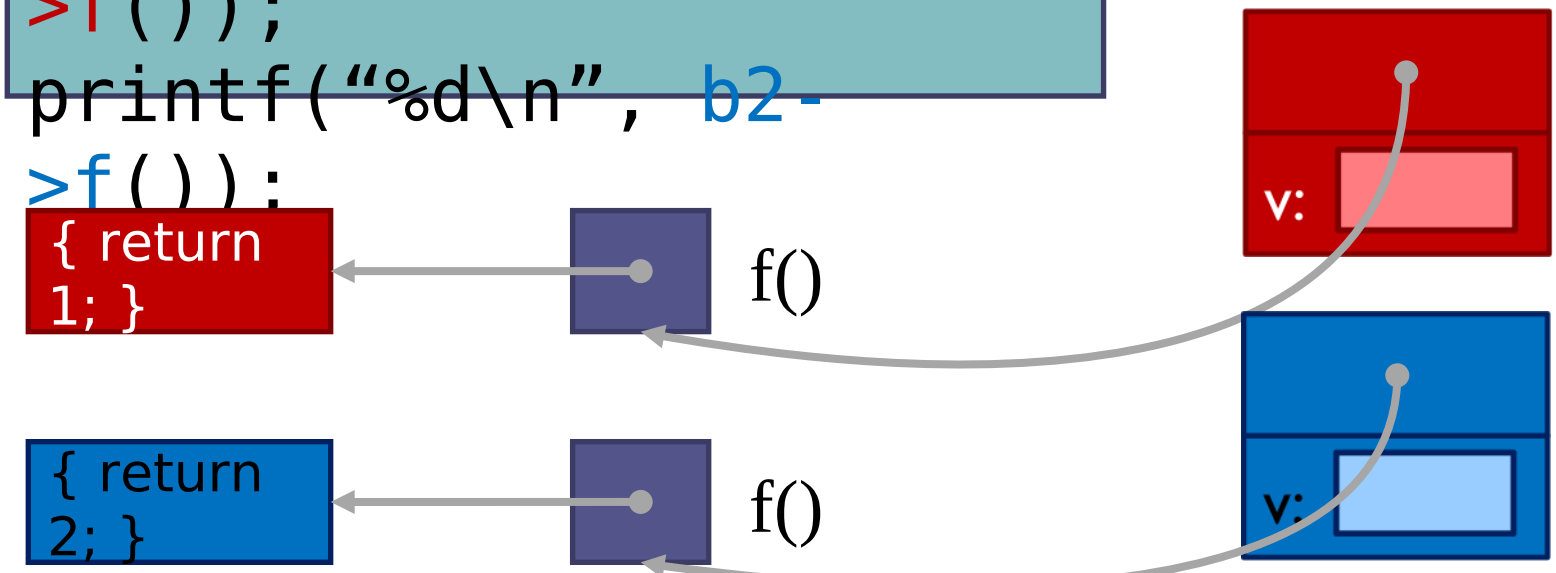
- Se esistono più istanze di una data classe, queste condividono la stessa V-Table




```
class CVirtual{
 protected:
      int i,j;
 public:
      virtual void foo() {...};
   virtual void bar() {...};
};
```

# Metodi virtuali e V-Table

```
Base* b1= new Der1();
Base* b2= new Der2();

printf("%d\n", b1->f());
printf("%d\n", b2->f());
```



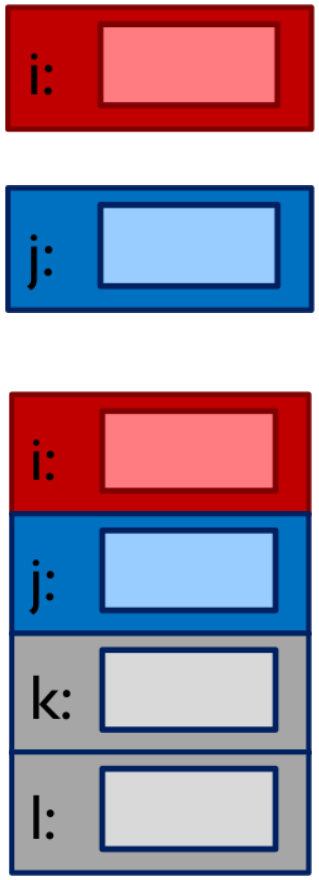
# Ereditarietà Multipla

- Una classe può ereditare da più di una classe
  - La sua interfaccia risulterà l'unione dei metodi contenuti nelle rispettive interfacce delle classi base uniti ai metodi propri della classe derivata
- Una classe può ereditare da più di una classe
  - Occorre evitare di derivare più volte dalla stessa classe



# Ereditarietà Multipla

```
cclass CBase1  {
  int i;
};
class CBase2 {
  int j;
};
class CDer:
    public CBase1,
    public CBase2
{  int k, l; };
```



# Operatori per Type Cast

- C++ supporta il type-cast in stile “C”
  - L’ereditarietà multipla e virtuale ne rendono pericoloso l’utilizzo
- C++ fornisce una serie di operatori per il type cast più efficienti e sicuri
  - static_cast<T>
  - dynamic_cast<T>
  - const_cast<T>
  - reinterpret_cast<T>



# static_cast<T>(p)

```
class Base1;
class Base2;
Class Derivata:
    public Base1,
    public Base2 { };
...
Derivata *d = new Derivata();
Base1 *b1;
Base2 *b2;

b1= static_cast<Base1 *>(d);
b2= static_cast<Base2 *>(d);
```

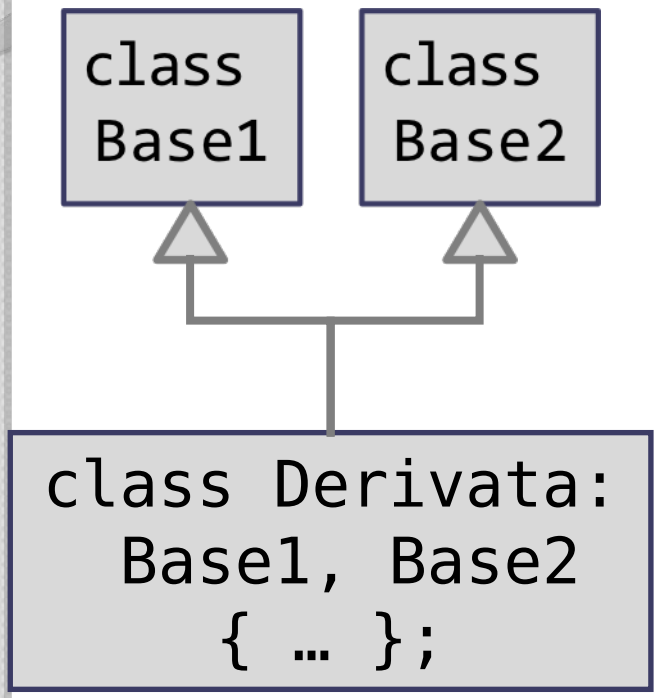


# static_cast<T>(p)

- Converte il valore di “p” rendendolo di tipo “T”
  - Se esiste un meccanismo di conversione disponibile
- Il meccanismo viene scelto in base al tipo di “p” come noto al compilatore
- Nel caso di puntatori, questo può essere diverso dal tipo effettivo di “p”
  - Non effettua controllo di compatibilità a run-time
- Se la conversione è illecita, il risultato non è predicibile
- Permette principalmente conversioni “in verticale” lungo l’asse ereditario
  - Sono possibili altre conversioni, se esistono espliciti operatori



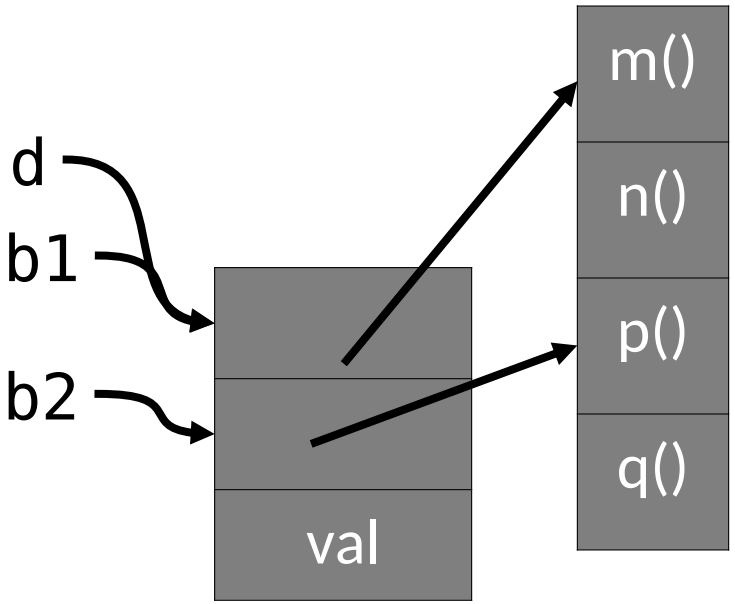
# static_cast<T>(p)




```
Derivata *d=new Derivata();
Base1 *b1;
Base2 *b2;

b1= static_cast<Base1 *>d;
b2= static_cast<Base2 *>d;
```

# static_cast<T>(p)

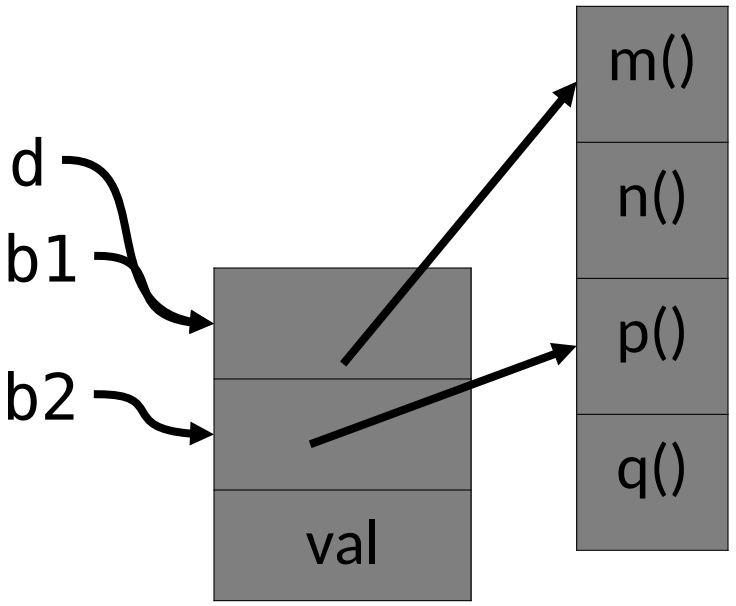
```
Derivata *d=new Derivata();
Base1 *b1;
Base2 *b2;

b1= static_cast<Base1 *>d;
b2= static_cast<Base2 *>d;
```




**Attenzione!** Anche se **b1==d** non è possibile scrivere
`b2=static_cast<base2*>(b1)`

# static_cast<T>(p)

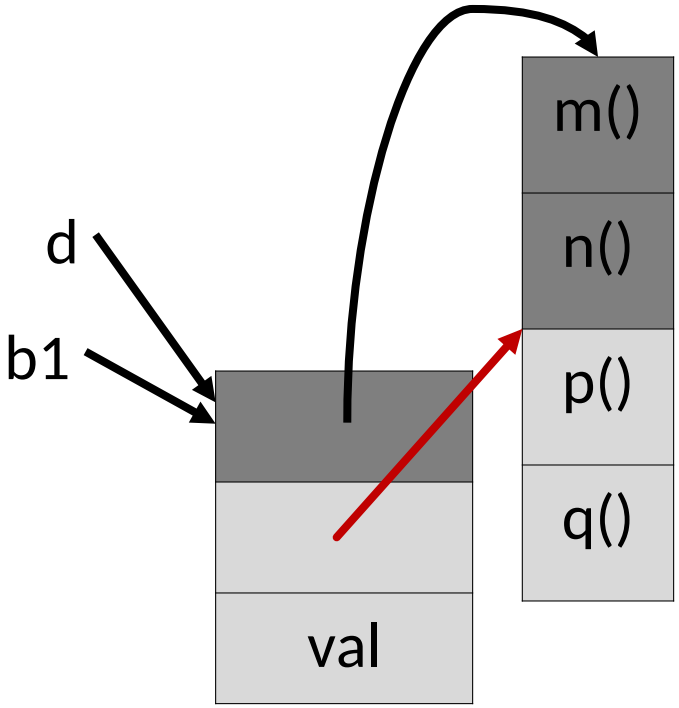
```
Derivata *d=new Derivata();
Base1 *b1;
Base2 *b2;

b1= static_cast<Base1 *>d;
b2= static_cast<Base2 *>d;
```




**Attenzione!** Anche se **b1==d** non è possibile scrivere
`b2=static_cast<base2*>(b1)`

# static_cast<T>(p)

```
Base1 *b1= new Base1();
Derivata *d;
Base2 *b2;

d=static_cast<Derivata*>b1;
```




Il compilatore non se ne accorge, ma in fase di esecuzione sorgono problemi

# dynamic_cast<T>(p)

- Effettua controllo di compatibilità run-time assicurando cast sicuri tra tipi di classi
- Applicato a un puntatore, ritorna 0 se il cast non è valido
- Applicato a un riferimento genera un'eccezione in caso di incompatibilità
- Può effettuare il "downcast" da una classe virtuale di base ad una derivata



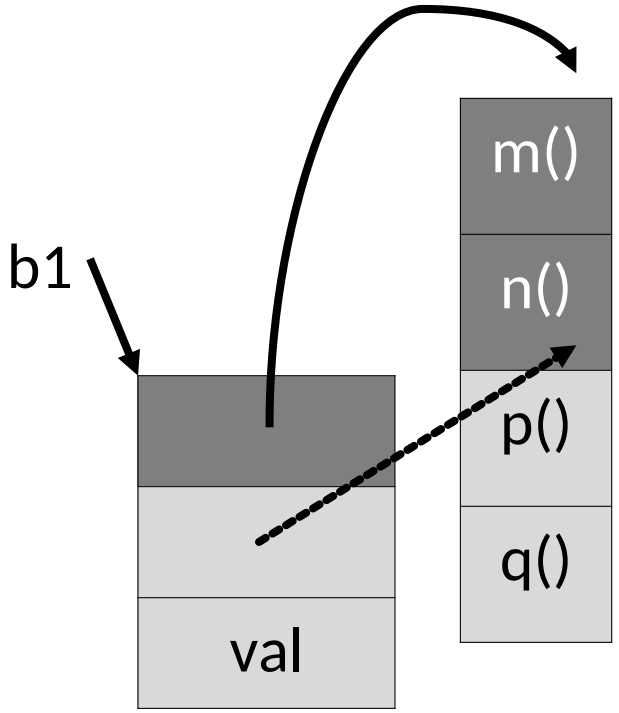
# dynamic_cast<T>(p)

```
class Base1;
class Base2;
Class Derivata:
    public Base1,
    public Base2 { };
...
Derivata *d = new Derivata();
Base1 *b1;
Base2 *b2;

b1= dynamic_cast<Base1 *>(d);
b2= dynamic_cast<Base2 *>(d);
```



# dynamic_cast<T>(p)

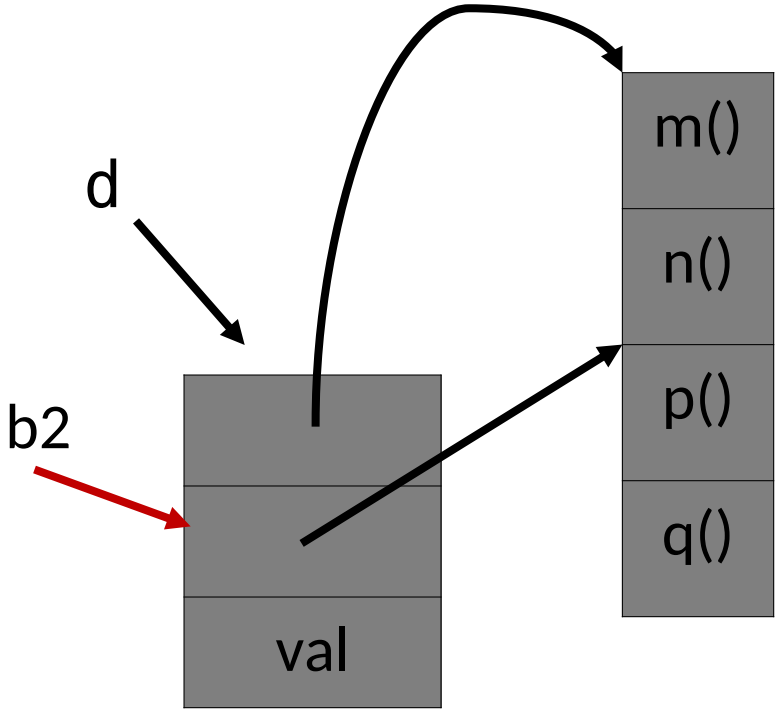
```
Base1 *b1 = new Base1();
Derivata *d;
Base2 *b2;

d = dynamic_cast< Derivata *>(b1);
```




Al contrario di static_cast, d = NULL

# dynamic_cast<T>(p)

```
Base2 *b2;
Derivata *d = new Derivata();
b2 = dynamic_cast<Base2*>(d);
```




In questo caso l'operazione non fallisce:
b2 punta alle interfacce di d, derivate dalla classe di base Base2

# reinterpret_cast<T>(p)

- Interpreta la sequenza di bit di un valore di un tipo come valore di un altro tipo
- Autorizza il compilatore a violare il sistema dei tipi
- Utilizzato di solito per dati ritornati da chiamate al sistema operativo o ricevuti dall'hardware
- Analogo al C-style cast



# const_cast<T>(p)

- Elimina la caratteristica di costante dal suo argomento
- Può generare problemi se usato su variabili globali poste dal compilatore in memoria di sola lettura



# Spunti di riflessione

- Si scriva una gerarchia di classi che estendono una classe base, definendo due metodi virtuali e si verifichi il comportamento polimorfico