



UNIVERSITÀ  
degli STUDI  
di CATANIA

# **Funzioni virtuali, polimorfismo, RTTI, Ereditarietà multipla**

Corso di programmazione I AA 2021/22

Corso di Laurea Triennale in Informatica

---

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: [gfarinella@dm.unict.it](mailto:gfarinella@dm.unict.it)

Dipartimento di Matematica e Informatica

Sia X una classe, Y una classe derivata **pubblicamente** da X:

- l'interfaccia di Y contiene, tra l'altro, l'interfaccia di X;
- Y si dice **sottotipo** della classe X;

### Principio di sostituzione di Liskov

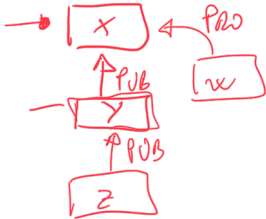
Un tipo S è un sottotipo di T quando è **possibile sostituire tutte le istanze di T con delle istanze di S** mantenendo **intatto il funzionamento del programma**.

In C++, un **puntatore** (risp. **reference**) ad un oggetto di tipo X può **puntare** a (risp. essere **alias** di)

- oggetti di tipo X
- istanze di **classi derivate direttamente o indirettamente** da X
- la **derivazione** (o la catena di derivazioni) deve usare lo specificatore **public**

## Puntatori vs gerarchie ereditarie.

```
1  class X { /* ... */ }
2  class Y: public X { /* ... */ }
3  class Z: public Y { /* ... */ }
4  class W: protected X { /* ... */ }
5  X x; Y y; Z z;
6  X *xptr = &x; // OK, tipo base
7  xptr = &y; // OK: X ← Y
8  xptr = &z; // OK: X ← Y ← Z
9  Y *yptr = &x; // Compile-time ERR!: X non è derivata di Y!
10 Y *yptr = &z; // OK: Y ← Z
11 W w;
12 X *wptr = &w; // Compile-time ERR!: W derivazione protected di X
```



### Reference vs gerarchie ereditarie.

```
1  class X { /* ... */ }
2  class Y: public X { /* ... */ }
3  class Z: public Y { /* ... */ }
4  class W: protected X { /* ... */ }
5  X x; Y y; Z z;
6  X &ref1 = x; // OK, tipo base
7  X &ref2 = y; // OK: X  $\leftarrow$  Y
8  X &ref3 = z; // OK: X  $\leftarrow$  Y  $\leftarrow$  Z
9  Y &ref4 = x; // Compile-time ERR!: X non è derivata di Y !
10 Y &ref5 = z; // OK: Y  $\leftarrow$  Z
11 W w;
12 X &wref = w; // Compile-time ERR!: W derivazione protected di X
```

## Puntatori e reference vs gerarchie ereditarie

Se ptr è un puntatore (risp. reference) di tipo A che punta a (risp. è alias di) un oggetto derivato pubblicamente da A, accesso diretto ad elementi specifici di B non sarà possibile senza opportuna conversione (si vedrà in seguito con RTTI).

```
1  class A{ ✓
2      public:
3          void foo() { /* ... */ }
4  };
5  class B: public A{
6      public:
7          void bar() { /* ... */ }
8  }
9  A* ptr = new B(); // OK
10 ptr->foo(); // OK
11 ptr->bar() // Compile-time error!
```

Esempi svolti

30\_01.cpp

## Polimorfismo

[...] assumere forme, aspetti, modi di essere diversi secondo le varie circostanze; possibilit di presentarsi in forme diverse[...] (fonte: Treccani)

- **interfaccia comune** in una gerarchia ereditaria, ovvero:
  - **gerarchia di classi** con metodi che hanno lo stesso nome; (**overriding**);
  - **differenti implementazioni** per lo stesso metodo all'interno della gerarchia;



# Polimorfismo a tempo di esecuzione

- puntatori (risp. reference) di un qualche tipo X che **puntano** a (risp. sono alias di) **oggetti di classi derivate da X** (direttamente o indirettamente);
- NB: A tempo di compilazione (compile-time) **non è noto il tipo di oggetto referenziato da puntatori e reference.**

```
1  class X { void foo(); };
2  class Y: public X { void foo(); };
3  class Z: public Y { void foo(); };
4  Y y, Z z;
5  X* ptr_y = &y; // puntatore a oggetto di tipo Y
6  X &ref_z = z; // puntatore a oggetto di tipo Z
7  ptr_y->foo(); // invocazione di X::foo()
8  ref_z.foo(); // invocazione di X::foo()
```



## Polimorfismo a tempo di esecuzione

Come ottenere il polimorfismo a tempo di esecuzione, ovvero il comportamento corrispondente all'oggetto selezionato durante l'esecuzione del programma?

Se le **funzioni della interfaccia** sono dichiarate `virtual`, si ottiene il **polimorfismo** a run-time.

```
1  class X { virtual void foo(); };
2  class Y: public X { void foo(); };
3  class Z: public Y { void foo(); };
4  Y y,    Z z;
5  X* ptr_y = &y; // puntatore a oggetto di tipo Y
6  X &ref_z = z; // puntatore a oggetto di tipo Z
7  ptr_y->foo(); // invocazione di Y::foo()
8  ref_z.foo(); // invocazione di Z::foo()
```

*ptr\_z ← y*

# Polimorfismo a tempo di esecuzione

Altro esempio.

```
1  class Cibo { public: virtual void cottura(); };
2  class Pasta:public Cibo { void cottura(); };
3  class Riso:public Cibo { void cottura(); };
4  Cibo *c;
5  cout << "Pasta o riso? (P/R)" << endl;
6  cin >> risposta;
7  if (risposta=="P")
8      c = new Pasta();
9  else
10     c = new Riso();
11  c->cottura();
```

## Polimorfismo a tempo di esecuzione

Ancora un altro esempio: chiamata “implicita” a metodo polimorfo.

```
1  class Cibo { public:  
2      virtual void cottura() { /* ... */ };  
3      void esegui_cottura() { /* ... */ cottura(); }  
4  class Pasta: public X { void cottura(); };  
5  class Riso: public Y { void cottura(); };  
6  Cibo *c;  
7  //...  
8  if (risposta=="P")  
9      c = new Pasta();  
10 else  
11     c = new Riso();  
12 c->esegui_cottura(); // Riso::cottura() o Pasta::cottura()
```

**Overriding + virtual = Polimorfismo dinamico** (a tempo di esecuzione).

- In presenza di puntatori e/o reference, al compilatore **non sarà noto il tipo di oggetto referenziato**.
- Il compilatore conserva informazioni (**tabella delle funzioni virtuali**) che permettono di identificare, a tempo di esecuzione il comportamento corrispondente all'oggetto referenziato;
- Tale meccanismo è denominato **late-binding** (binding dinamico)

### Overloading = Polimorfismo statico.

- La risoluzione avviene a tempo di compilazione.
- Il compilatore applica un insieme di **regole di risoluzione dello overloading** per determinare, a tempo di compilazione, la versione corrispondente all'invocazione.

### Remarks

Una classe **polimorfa** definisce una **interfaccia** “comune” a differenti implementazioni.

Le differenti implementazioni sono presenti in **classi derivate** (direttamente o indirettamente) della classe polimorfa (gerarchia ereditaria).

La “catena” di **derivazioni** deve essere **pubblica**.

Le funzioni della interfaccia debbono essere dichiarate **virtuali**.

### Remarks (cont.)

Il polimorfismo a tempo di esecuzione avviene mediante **puntatori e/o reference**:

- in assenza di puntatori o reference non vi è la necessità di “risolvere” le chiamate, in quanto il tipo è noto.



### Esempi svolti

30\_02.cpp // gerarchia ereditaria, overriding di funzioni non virtuali

30\_03.cpp // gerarchia ereditaria, overriding di funzioni virtuali

**Un costruttore non può essere dichiarato virtuale.**

**Costruzione/allocazione** di oggetto (automatica o dinamica) di classe derivata implica (bottom up):

1. invocazione costruttore classe base
2. invocazione costruttore classe derivata

Distruzione oggetto allocato sullo stack (**il tipo e' noto**) implica (top-down)

1. invocazione distruttore classe derivata
2. invocazione distruttore classe base

Deallocazione oggetto allocato sullo heap (alloc. dinamica) implica:

1. invocazione distruttore classe derivata **SOLO** se **il distruttore della classe base è stato dichiarato virtual;**
2. invocazione distruttore classe base

## Esempi svolti

30\_04.cpp // polimorfismo vs costruttori e distruttori

30\_05.cpp // collezioni di oggetti differenti

30\_06.cpp // collezioni di oggetti differenti

## Metodi virtuali puri, classi astratte, interfacce

Un concetto di base può essere per certi aspetti molto vago.

In termini pratici, una certa classe potrebbe definire uno o più metodi (esempio: `perimetro()` per la classe poligono per i quali tuttavia **non è possibile fornire una implementazione di base.**

SOLUZIONE: Funzione virtuale pura.

```
1  class Figura{  
2      virtual void area() = 0; //nessuna implementazione  
3  }
```

# Metodi virtuali puri, classi astratte, interfacce

```
1  class Figura{  
2      virtual void area() = 0; //nessuna implementazione  
3  }
```

- Una classe con una o più funzioni virtuali pure si dice **classe astratta**
- **Non si possono creare istanze** di una classe astratta.
- È comunque possibile dichiarare **puntatori e reference a tipi astratti**.
- Se una classe derivata **non fornisce l'implementazione** di una o più **funzioni virtuali pure**, allora **anche essa diverrà astratta**;

### REMARKS

Una funzione membro di una classe può essere dichiarata:

- **senza lo specificatore** `virtual`: in questo caso, dalla ridefinizione della funzione membro in una classe derivata, NON si avrà polimorfismo a run-time (**Hiding**).
- **con lo specificatore** `virtual`: in questo caso, la ridefinizione della funzione membro permetterà polimorfismo a run-time (**Overriding**).



- **con lo specificatore** `virtual` ed inoltre **non essere definita**: in questo caso, la classe derivata DEVE ridefinire la funzione membro. In caso contrario anche essa sarà astratta.

## Interfaccia

Una classe base astratta che contiene solamente i) attributi costanti e ii) funzioni virtuali pure si dice **Interfaccia**.

In un interfaccia nessun comportamento è definito, ma semplicemente dichiarato (solo specifiche, senza implementazione).

Non si confonda il concetto di interfaccia con il termine *interfaccia* riferito all'insieme dei metodi pubblici di una classe.

È noto che un puntatore (risp. reference) ad una certa classe può puntare a (risp. essere alias di) un oggetto una classe derivata.

Il polimorfismo a tempo di esecuzione si occupa della selezione della versione corretta delle funzioni membro (virtuali) ridefinite nella gerarchia ereditaria.

Tuttavia, si potrebbe avere la necessità di conoscere il tipo dell'oggetto referenziato per sfruttare delle funzionalità specifiche.

Dunque: Come stabilire il tipo dell'oggetto referenziato a tempo di esecuzione?

# RTTI: Run-Time Type Identification

Il problema è di passare da un tipo ad un altro appartenente alla stessa gerarchia ereditaria mediante conversione del puntatore

Prima non-soluzione (ptr potrebbe puntare ad un oggetto di tipo B.): tentare una conversione esplicita (type-cast) *C-style*.

```
1  class A { /* ... */ }
2  class B: public A { void b(); }
3  class C: public A { void c(); }
4  //..
5  B b; C c;
6  A *ptr = &b;
7  //..
8  ((C *) ptr) -> c(); // 1. C-style cast, NON safe!
9  //..
```

Altra non-soluzione: `static_cast`.

Operatore introdotto nel C++ (si dice *named cast*), al fine di rendere più esplicite operazioni di conversione tra tipi “compatibili”:

- due classi appartenenti alla stessa gerarchia;
- intero a numero in virgola mobile, ...
- conversioni mediante costruttori oppure operatori di conversione;

# RTTI: Run-Time Type Identification

```
1  class A { /* ... */ }
2  class B: public A { void b(); }
3  class C: public A { void c(); }
4  //..
5  B b; C c;
6  A *ptr = &b;
7  //..
8  C* ptr_c = static_cast<C*>(ptr); //C++
9  ptr_c->c(); // NOT safe!
```

## Esempi svolti

30\_08.cpp – C-style cast e static\_cast

# RTTI: Run-Time Type Identification

**Prima soluzione: operatore `dynamic_cast`** controlla il tipo dell'oggetto referenziato dal puntatore a tempo di esecuzione

- Se del tipo richiesto, restituisce il puntatore stesso.
- Altrimenti restituisce `nullptr`.

```
1  class A { /* ... */ }
2  class B: public A { void b(); }
3  class C: public A { void c(); } ←
4  B b; C c;
```

```
5  A *ptr = &b;
```

```
6  C* ptr_c = dynamic_cast<C*>(ptr) -> c(); ←
```


```
7  if(ptr_c) ←
```

```
8  ptr_c -> c();
```



# RTTI: Run-Time Type Identification

Altra soluzione: usare operatore **typeid()**.

```
1  #include <typeinfo>
2  class A { /* ... */ }
3  class B: public A { void b(); }
4  class C: public A { void c(); }
5  B b; C c;
6  A *ptr = &b;
7  if (typeid(*ptr) == typeid(C)) 
8      ((C*) ptr->c) -> c();
```

## Esempi svolti

30\_09.cpp – operatore `dynamic_cast`

30\_10.cpp – operatore `typeid`


Una classe può derivare contemporaneamente più di una classe base.

ES:

- Classe **studente**
- Classe **atleta**.
- Squadra universitaria di atleti: collezione di oggetti di tipo **AtletaUniversitario**, che necessita delle caratteristiche di entrambe le classi.

# Ereditarietà multipla

La sintassi di una derivazione multipla prevede di inserire **uno specificatore di accesso per ogni superclasse**.

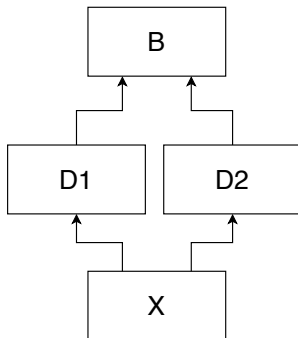


```
1 x class A { float a; /* ... */ };
2 x class B { string b; /* ... */ };
3 x class C: public A, public B { /* ... */ };
4 //...
5 C c = new C(); // object C has fields a and b..
```

## Ereditarietà multipla

Uno scenario di questo tipo genera **ambiguità** nella risoluzione dei nomi (attributi) ereditati dalla classe base.

**Duplicazione di memoria.**



# Ereditarietà multipla

```
1  → class B { int x /* ... */ };
2  → class D1 : public B { /* ... */ };
3  → class D2 : public B { /* ... */ };
4  → class X: public D1, public D2 {
5      void foo() { x=10; } }; /*ambigua! Compile-time ERR*/
```

in quanto attributo x definito in B viene ereditato **due volte**.

```
1      class B { int x /* ... */ };
2      class D1 : public B { /* ... */ };
3      class D2 : public B { /* ... */ };
4      class X: public D1, public D2 {
5          void foo() { D1::x = 10; D2::x = 20; }; / * OK */
```

## Esempi svolti

30\_11.cpp – ereditarietà multipla, ambiguità e duplicazione di memoria

## Ereditarietà multipla

Per ovviare al problema si può utilizzare lo specificatore `virtual` nella derivazione diretta dalla classe base.

```
1      class B { int x /* ... */ };
2      class D1 : virtual public B { /* ... */ };
3      class D2 : virtual public B { /* ... */ };
4      class X: public D1, public D2 {
5          void foo(){ x = 10; } / * OK */ };
```

In questo modo l'attributo **x** sarà **ereditato solo una volta** dalla classe **X**.



## Esempi svolti

30\_12.cpp – ereditarietà multipla, classi virtuali

**FINE**