# Alcune questioni tecniche sul polimorfismo dinamico

Abbiamo visto che, nel caso di polimorfismo dinamico, le classi astratte sono tipicamente formate da metodi virtuali puri, più il distruttore della classe che è dichiarato virtuale, ma non puro. In alcuni casi è però necessario complicare il progetto (ad esempio, usando l'ereditarietà multipla): quando lo si fa, si corre il rischio di incorrere in errori ed è quindi opportuno cercare le risposte ad alcune domande tecniche sul polimorfismo dinamico, che possono diventare rilevanti quando viene utilizzato al di fuori dei confini stabiliti nei nostri semplici esempi.

1. Ci sono metodi che NON possono essere dichiarati virtuali? In particolare, cosa si può dire sulla possibilità o meno di rendere virtuali le seguenti categorie di metodi:
    - costruttori
    - distruttori
    - funzioni membro (di istanza, cioè non statiche)
    - funzioni membro statiche
    - template di funzioni membro (non statiche)
    - funzioni membro (non statiche) di classi templatiche
2. Come faccio a costruire una copia di un oggetto concreto quando questo mi viene fornito come puntatore/riferimento alla classe base?
3. Cosa succede se si invoca un metodo virtuale durante la fase di costruzione o di distruzione di un oggetto?
4. Come funziona l'ereditarietà multipla quando NON ci si limita al caso delle interfacce astratte?
    - scope e ambiguità

    - `classi base ripetute`

    - `classi base virtuali`

    - `semantica speciale di inizializzazione`

5. Quali sono gli usi del polimorfismo dinamico nella libreria standard?
    - classi eccezione standard

    - `classi iostream`

---

# Metodi che non possono essere `virtual`

## Costruttori (NO)

Al momento della creazione, non esiste ancora un oggetto `this` interrogabile (esiste ma lo si sta definendo in quel momento).

## Distruttori (SI, tipicamente obbligatorio)

Vogliamo eliminare correttamente l'oggetto nella sua interezza; inoltre deve essere implementato in quanto verrà sicuramente invocato nella catena di chiamate ai distruttori (dall'oggetto di tipo `Derived` a quello `Base`).

```
struct A {
 ~A() { }
};

struct B : public A {
 ~B() { }
};

{
 A* pa_b = new B;
 delete pa_b; // errore: non chiama ~B()
}
```

## Funzioni membro (SI)

Sono le classiche funzioni membro di una classe standard.

### Funzioni membro statiche (NO)

Sono funzioni che fanno riferimento alla classe, non hanno il puntatore `this`, non c'è nessun modo che permetta al RTTI (*Run Time Type Identification*) di effettuare una qualche forma di dispatching della funzione in questione.

### Template di funzioni membro, non statiche (NO)

Il `this` esiste, quindi in linea teorica sarebbe legittimo dichiararle virtuali; si è deciso però di non implementare questa possibilità, per efficienza. Facciamo un esempio:

```
struct F {
 virtual void do() = 0; // OK

 template
 virtual void foo() = 0; // Errore
};
```

Così facendo, l'utente potrebbe istanziare un numero grande a piacere di funzioni virtuali `foo()`, il che significherebbe una grandezza arbitraria della **V-Table** (tabella dei metodi virtuali).

### Funzioni membro (non statiche) di classi templatiche (SI)

Una volta scelto il tipo templatico della classe in questione, l'insieme delle funzioni virtuali è finito e numerabile, quindi è ammissibile:

```
template
class Animale {
public:
 virtual void verso() = 0;
 virtual void foo(T& t) = 0;
};

// Animale a; --> ci sono 2 funzioni virtuali
```

---

# Copia di un oggetto concreto

Intuitivamente serve un metodo virtuale che costruisca correttamente un nuovo oggetto a partire dal riferimento/puntatore passato: il metodo `clone()`.

```
class Animale {
 virtual void verso() const = 0;
 virtual Animale* clone() const = 0;
};

class Cane : public Animale {
 void verso override() const { /*  */ }
 Cane* clone() const override { return new Cane(*this); }
};

void foo(const Animale* pa) {
 Animale* pa_2 = pa->clone();
 // ...
}
```

Osserviamo che nell'implementazione del metodo di clonazione, il tipo restituito può essere cambiato con quello della classe derivata (`Cane*` invece di `Animale*`, eccezione del linguaggio), evitando *down-casting* esplicito. Questi metodi vengono detti costruttori virtuali.

---

# Invocazione di funzione virtuale in un costruttore/distruttore

La risoluzione dell'overriding in questo caso specifico funziona in maniera differente. Fino al `c++98` si adottava l'approccio classico, ed era possibile incorrere in evidenti problematiche: al momento della costruzione di un oggetto Derived, se il costruttore di Base facesse una chiamata ad un metodo virtuale, ci sarebbe accesso a memoria non ancora

inizializzata e possibile *undefined behaviour*:

```
struct Base {
 Base() {
  foo();
 }
 virtual void foo () { std::cout << "Base::foo()" << std::endl; }
};

struct Derived : public Base {
 int* p_i;

 Derived() : p_i(new int) {
  *p_i = 7;
  foo();
 }
 virtual ~Derived() { delete p_i; }
 virtual void foo() { std::cout << *p_i << std::endl; }
};

{
 Derived d;
}
```

Il costruttore di `Derived` chiama implicitamente quello di `Base`, la chiamata a `foo()` viene risolta con `Derived::foo()` andando quindi ad effettuare una dereferenziazione di un puntatore non ancora inizializzato.

Dallo standard `c++11` il supporto a tempo di esecuzione cambia incrementalmente il tipo dinamico dell'oggetto in questione: al momento della costruzione della parte `Base` di un oggetto di tipo `Derived`, quello stesso oggetto viene considerato di tipo `Base` e viene invocata correttamente `Base::foo()`; a questo punto, il tipo dinamico dell'oggetto cambia in `Derived` e viene invocata `Derived::foo()`.

Stampa:

```
Base::foo();
7
```

Nei distruttori si ha lo stesso comportamento.

---

# Ereditarietà multipla con classi non interfaccia

Utilizzando più classi base con lo stesso campo, potrebbero esserci oggetti ripetuti:

```
struct B1 {
 int a;
};
struct B2 {
 int a;
};

struct D : public B1, public B2 {
 void foo() {
  std::cout << a << std::endl; // ambiguo, dovrei specificare B1::a o B2::a
 }
};
```

Supponiamo ora un altro caso:

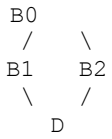
```
struct B0 {
 int a;
};
struct B1 : public B0 {
};
struct B2 : public B0 {
};

struct D : public B1, public B2 {
};
```

È possibile avere una struct `B0` condivisa tra `B1` e `B2` nella struct `D` ? Si, utilizzando l'ereditarietà di tipo `virtual`.

```
struct B1 : virtual public B0 {
};
struct B2 : virtual public B0 {
};
```

Si genera uno schema *a diamante* in cui la classe `D` deriva da altre due classi che a loro volta derivano comunemente da `B0`.

> Conosciuto come *DDD* problem (Dreadful Diamond on Derivation)




Nella costruzione di un oggetto di tipo `D`, la semantica di costruzione è diversa: prima le classi base virtuali (in questo caso `B0`) nell'ordine di visita del grafo di ereditarietà, successivamente tutte le classi base che non sono virtuali (`B1` e `B2`) evitando di re-inizializzare quelle virtuali; infine si procede con la costruzione dell'oggetto derivato.

La stessa semantica è applicata nella distruzione, in ordine inverso: distruggo l'oggetto derivato, successivamente `B1` e `B2` senza toccare `B0` e infine il distruttore di `D` si occupa di chiamare quello di `B0`.