**∎** Sia D classe che deriva da B e C in cui sono presenti due metodi distinti con lo stesso nome m(), cioè si hanno B::m() e C::m()

⇒ Se su D si prova ad invocare m() si causa un errore di compilazione causato dall’ambiguità, per risolvere si deve usare lo scooping.

∎ **Costruttore per D** :

1. Viene invocato il costruttore della classe base B da cui deriva, esplicitamente o implicitamente;

2. Viene eseguito il costruttore per i campi dati propri di D;

3. Viene eseguito il corpo del costruttore di D(standard o ridefinito).

∎ **Costruttore di Copia Standard per D** invocato su un oggetto x di D:

1. Costruisce il sottooggetto B di D invocando il costruttore di copia B(const B&) standard o ridefinito;

2. Costruisce ordinatamente i campi dati propri di D invocando i relativi costruttori di copia.

∎ **Costruttore di Copia ridefinito** in D, esso può invocare esplicitamente il costruttore di copia di B o qualsiasi altro costruttore di B.

•Se non ci sono invocazioni esplicite per il costruttore di copia di B ⇒ Viene invocato automaticamente il costruttore di default di B e

quindi non il costruttore di copia di B.

**∎ Assegnazione Standard** per D:

1. Invoca l’assegnazione della classe base B(standard o ridef) sul sottooggetto corrispondente;

2. Esegue l’assegnazione ordinatamente membro per membro dei campi dati propri di D, invocando le corrispondenti assegnazioni(standard o ridef).

**∎ Assegnazione ridefinita** in D: 1. Viene eseguito solo il suo corpo, la ridefinizione dell’assegnazione non provoca nessuna invocazione implicita.

∎ **Distruttore Standard** per D:

1. Invoca il distruttore standard proprio di D ⇒ distrutte i campi dati propri di D nell’ordine inverso a quello di costruzione tramite i corrispondenti

distruttori(standard o ridef);

2. Invoca implicitamente il distruttore delle classi B da cui deriva(standard o ridef) per distruggere i sottooggetti B;

∎ **Distruttore ridefinito in D** : 1. Viene eseguito il corpo del distruttore di D;

2. Viene invocato il distruttore delle classi base B(standard o ridef) per distruggere i sottooggetti di B.

∎ **Metodo Virtuale**: virtual void m(); ⇒ fa in modo che venga invocato il metodo m() della classe del tipo dinamico del punt / rif. solo se questo fa

parte della gerarchia.

• Quando m() è dichiarato virtual ⇒ è virtuale in tutta la gerarchia delle classi che derivano da B.

• Overriding di m() : quando ridefiniamo un metodo virtual m() di una classe base B su una sua sottoclasse D (deve mantenere la stessa segnatura);

• Si può **cambiare tipo di ritorno su m() solo se il tipo di ritorno è un punt / rif ad una sottoclasse del tipo della classe di m() di B**;

• Se m() ha valori di default su B, cioè m(int x = 0) ⇒ nell’override di m() su una classe derivata D i valori di default possono essere omessi, deve

comunque mantenere la stessa segnatura m(int x).

• Affinchè un invocazione del metodo virtuale m() tramite puntatore p di tipo B\* possa compilare ⇒ m() deve essere disponibile nella classe B.

∎ Sia B sottotipo di A, **B\* pb = new B;** e **A\* pa1 = pb;** ⇒ **pa1 -> f(1);**  NB: f(int) ≠ f(const int)

• Se in A **f(int) è virtuale** ⇒ • Se f(int) è ridefinita in B e TD(pa1) = B ⇒ esegue B::f(int)

• Se f(int) non è ridefinita in b e TD(pa1) = B ⇒ esegue A::f(int)

• Se TD(pa1) = A ⇒ esegue A::f(int)

• Se in A **f(int) non è virtuale** ⇒ • Qualsiasi sia il TD(A) ⇒ esegue A::f(int)

∎ **Distruttori Virtuali**: nel caso in cui facessimo una delete su un punt / rif il cui TD ≠ TS, per fare in modo che venga chiamato il distruttore della

classe del TD bisogna che il distruttore della classe derivata da D venga dichiarato virtuale nella classe base B ⇒ In questo modo tutti i

distruttori della gerarchia diverranno virtuali.

• Di solito si dichiara virtuale il distruttore della classe base B ma con il corpo vuoto, anche se non è strettamente necessario

⇒ in questo modo viene invocato il distruttore standard per B e tutte le classi che derivano da essa.

∎ **Metodi Virtuali Puri**: serve per dichiarare classi base B “astratte” in modo da essere utilizzate solo per rendere le classi derivate D “concrete”.

Su B ci si limita a specificare le operazioni basilari che tutte le classi derivate D dovrebbero avere.

• virtual void G() = 0; ⇒ Una classe astratta diventa concreta solo se nella classe derivata D si fa l’override di G() senza dichiararlo puro di nuovo.

∎ **Derivazione Virtuale**: nel caso di ereditarietà a diamante si producono ambiguità a spreco di memoria. Per evitare ciò ed avere un solo oggetto

della classe B nel livello più basso della gerarchia a diamante bisogna dichiarare virtuale la derivazione delle due classi al centro della gerarchia.

• In presenza di classi base B virtuali, il **costruttore di una classe derivata D oltre a richiamare preliminarmente i costruttori delle classi dalle quali deriva direttamente,**

**richiama inoltre preliminarmente anche i costruttori delle classi virtuali presenti nella sua gerarchia di derivazione**.

Ciò è necessario dato che il sottooggetto di una classe base virtuale A è condiviso, e non avrebbe senso costruirlo più di una volta.

⇒ Il sottooggetto della classe base virtuale A va costruito prima che avvenga la sua condivisione.

∎ **Costruttore di una classe derivata D con ereditarietà multipla e classi virtuali**:

1. Vengono richiamati, una sola volta, i costruttori delle classi virtuali presenti nella gerarchia di derivazione di D:

⇒ Se vi è più di una classe virtuale in D:

1. La ricerca delle classi virtuali procede esaminando per prime le superclassi dirette di D in ordine di dichiarazione di

ereditarietà( da sx verso dx nella gerarchia);

2. Per ognuna di queste sottogerarchie si cercano ricorsivamente le classi base virtuali.

2. Vengono richiamati i costruttori delle superclassi dirette non virtuali di D;

3. Viene eseguito il costruttore proprio di D : 1. Campi propri; 2. Corpo del costruttore di D.

•Nel caso di gerarchia a diamante, il compito di costruire il sottooggetto della classe base virtuale A è lasciato alla classe dell’oggetto principale che stiamo definendo, nel

nostro caso al costruttore di D.

Perciò una eventuale chiamata esplicita al costruttore della classe virtuale A deve essere messa nella lista di inizializzazione di D.

•Se nelle liste di inizializzazione delle classi B e C compare un’invocazione del costruttore di A ⇒ nella costruzione di un oggetto di D tale chiamata verrà ignorata.

∎ **Distruttore di una classe derivata D con ereditarietà multipla e classi virtuali**:

• Se ho **B\* pb = new B;** e **A\* pa1 = pb;** ⇒ con **delete pa1;** ⇒ richiama solo il distruttore di A anche se TD(pa1) = B

⇒ con **delete pb;** ⇒ richiama i distruttori partendo dal basso della gerarchia(fa il contrario del costruttore)

• Un **tipo T è polimorfo se è un tipo classe che include almeno un metodo virtuale**;

• Una **classe polimorfa C è una classe che contiene solo il distruttore virtuale**;

• Ogni sottotipo di un tipo polimorfo è a sua volta un tipo polimorfo ⇒ quindi **ogni classe derivata da una classe polimorfa è polimorfa**.

∎ **Typeid** : permette di determinare il TD di una qualsiasi espressione(punt o rif) a run-time

• **typed(\*p)** ⇒ • Ritorna il TD se il tipo del punt/rif è di una classe polimorfa(se lo uso su un puntatore non dereferenziato ritorna il TS);

• Ritorna il TS se non è polimorfa.

( Il const viene ignorato )

∎ **Dynamic** **Cast**: **dynamic\_cast<D\*>(\*p)** ⇒ converte il puntatore p nel tipo target D solo se il tipo di p è classe polimorfa B dalla quale D deriva;

Perciò se TD(p) = E\* ⇒ • Se E è sottotipo di D ⇒ OK la conversione va a buon fine;

• Se E non è sottotipo di D ⇒ ritorna puntatore o riferimento nullo.

• **Safe Downcast** : quando si converte dall’alto verso il basso nella gerarchia tramite **dynamic\_cast<D\*>(p)** , cioè da classe base B a derivata D: **B\* ⇒ D\*** || **B& ⇒ D&**

**• Up Cast:** quando si converte dal basso verso l’alto tramite un assegnazione: **D d; B\* b = d;** cioè da classe derivata D alla classe base B: **D\* ⇒ B\*** || **D& ⇒ B&**

**∎ const\_cast<T\*>(\*p) :** converte p puntatore o riferimento costante a T in un punt/rif non costante T.

∎ Un **metodo static** si usa quando l’azione del metodo è indipendente dall’oggetto di invocazione, cioè quando l’**oggetto di invocazione non è necessario per definire il**

**metodo** e perciò non possono essere dichiarati const. Non hanno parametro implicito this.

Es: static orario OraPranzo(){ return orario(13, 14); } ⇒ cout<<”orario::OraPranzo().Ore()”<<”orario::OraPranzo().Minuti()”;

JZ