6. Polimorfismo Dinamico

In questo capitolo vedremo uno dei pilastri, assieme a incapsulamento e ereditarietá, della programmazione orientata agli oggetti.
Con polimorfismo si intende la capacitá di fornire una singola interfaccia ad entitá di diversi tipi al fine di rendere il codice piú versatile.

Relazione IS-A

Come giá sappiamo la derivazione pubblica consente di effettuare, in maniera implicita, conversioni di tipo **up-cast**.
Conversioni da un puntatore (o riferimento) per un oggetto della classe figlia verso un puntatore (o riferimento) per un oggetto della super classe.
Vediamo quindi un banale esempio.

```
// Super classe.
class Base {
     /* ...  */
};

// Classe figlia.
class Derived :  public Base {
     /* ...  */
};
```

La conversione concessa di cui parlavamo prima, in questo caso, sarebbe:

```
Base* base_ptr = new Derived;
```

Questa conversione indica che é possibile utilizzare un oggetto `Derived` (tipo concreto) come se fosse un oggetto `Base` (tipo astratto), ignorando le eventuali caratteristiche specifiche di `Derived` e concentrandosi su quelle in comune con `Base`.

Si dice quindi che la classe `Derived` é in relazione **IS-A** con la classe `Base`.

Questo vuol dire che rappresenta una particolare concretizzazione della classe `Base` e puó quindi essere usato come se fosse un oggetto di tipo `Base`.

Metodi Virtuali

Sono metodi definiti in classi derivate che possono essere interrogati a tempo di esecuzione per ridirezionare dinamicamente la chiamata del metodo alla sua classe corretta.

Cerchiamo di capire questo concetto vedendo il seguente codice di esempio:

```
class Stampante {

      public:
            void stampa(const Doc& doc);
};

class File_Stampante :  public Stampante {

      public:
            void stampa(const Doc& doc);
};

class Rete_Stampante :  public Printer {

      public:
            void stampa(const Doc& doc);
};
```

Supponiamo ora che il codice dell'utente voglia stampare alcuni documenti utilizzando una stampante e, non essendo interessato a dettagli implementativi, utilizza l'astrazione di `Stampante` nel modo seguente:

```
void stampa_tutti(const std::vector<Doc>& docs, Stampante* printer)
{
      for(const auto& doc :  docs)
            printer->stampa(doc);
}
```

In questo caso, il chiamante invoca la funzione `stampa_tutti` passando un puntatore a una stampante concreta (`File_Stampante` o `Rete_Stampante`) sfruttando l'up-cast.
Il compilatore, quindi, eseguirá l'overloading effettuando la ricerca nella classe `Stampante` (senza cercare nelle classi derivate) e troverá solamente il merodo `Stampante::stampa`.

L'utente, peró, intendeva invocare il metodo specifico della stampante concreta passata alla funzione.

Infatti, ogni classe concreta ridefinisce il metodo `stampa` a seconda delle necessitá della classe di appartenenza (ad esempio `Rete_Stampante` potrebbe tenere traccia del numero di pagine stampate da vari utenti).

Serve un meccanismo che consenta di interrogare a run-time il puntatore allo scopo di capire il suo tipo dinamico e poterlo cosí ridirezionare alla classe corretta.

Questo meccanismo viene detto **RTTI** ovvero Run-Time Type Identification.
Effettua dunque la risoluzione dell'overridng e puó essere attivato solo se i metodi della classe sono **metodi virtuali**.
Una classe che contiene almeno un metodo virtuale viene detta **classe dinamica**.

```
class Stampante {

     public:
          virtual void stampa(const Doc& doc);
};
```

Metodi Virtuali Puri

Quando si definisce una classe base (come `Stampante`) spesso non si ha la possibilitá di fornire un'implementazione sensata dei metodi virtuali.
Questo perché la classe fornisce solo l'interfaccia del concetto astratto di una stampante.

Per questo é preferibile indicare il metodo virtuale come **puro** usando la sintassi "`= 0`" al termine della dichiarazione.

```
class Stampante {

     public:
          virtual void stampa(const Doc& doc) = 0;
};
```

Una classe che contiene solo metodi virtuali puri viene detta **classe astratta**.
Non si possono dichiarare oggetti che abbiano come tipo una classe astratta: possono essere solo usate come classi base per altre classi derivate.

Distruttori Classi Astratte

I distruttori per le classi astratte devono essere dichiarati `virtual` e non devono mai essere puri.

```
virtual ~Classe_Astratta() {}
```

La ragione é di consentire la giusta distruzione degli oggetti delle classi concrete derivate da quella astratta.

Se il distruttore non fosse virtuale si verificherebbe memory leak.
Ad esempio:

```
class Astratta {

      public:
            virual void print() const = 0;
            ~Astratta() {} // Distruttore errato:  non é virtual.
};

class Concreta :  public Astratta {

      std::vector<std::string> vs;

      public:
            Concreta() :  vs(20, "stringa") {}

            void print() const override {
                  for(const auto& s :  vs)
                        std::cout << s << std::endl;
            }
            // Il distruttore di default andrebbe bene:  lo ridefiniamo
            // solo per fargli stampare qualcosa e osservare quindi
            // che non viene invocato.

            ~Concreta() { std::cout << "Distruzione" << std::endl; }
}

int main() {

      Astratta* a = new Concreta;
      a -> print();

      // Memory Leak!  non viene distrutto il vettore nella classe concreta.
      delete a; // Invoca il distuttore di Astratta che non é virtual.
}
```

Risoluzione Overriding

Viene effettuata a tempo di esecuzione dal **RTS** (Run-Time Support) del linguaggio.
Affinché si attivi questo meccanismo occore che:

1. Il metodo invocato sia virtuale.

2. Il metodo viene invocato tramite puntatore o riferimento.
   Altrimenti non c'é distinzione tra il tipo statico e il tipo dinamico e si invoca il metodo della classe base.

3. Almeno una delle classi della catena di derivazione che porta dal tipo statico al tipo dinamico ha giá effettuato overriding.

4. Il metodo NON deve essere invocato mediante qualificazione esplicita.
   Questo causerebbe l'invocazione del metodo cosí come é stato definito nella classe usata per la qualificazione.

Conversioni Esplicite in C++

C++ offre varie sintassi per effettuare il cast (conversione esplicita di tipo) di un'espressione allo scopo di ottenere un tipo di dato diverso.

Prima di vedere le varie tipologie e sintassi di cast, vediamo le motivazioni per cui diventa necessario effettuare una conversione esplicita:

1. Il cast implementa una conversione di tipo non consentita dalle regole del linguaggio.
   Il programmatore si prende la responsabilitá della correttezza della conversione.

2. Analogo al caso precedente ma in questo caso viene usato un `dynamic_cast` per controllare, a run-time, se la conversione richiesta é effettivamente consentita.

3. Viene inserito nel programma a scopo di documentazione.
   Non é necessario dal punto di vista tecnico, ma migliora la leggibilitá del codice.

4. Conversione forzata al tipo `void` che corrisponde a una richiesta di ignorare il valore dell'espressione.

Descriviamo ora le diverse tipologie di cast.

- **Static**
  Calcola un nuovo valore ottenuto dalla conversione dell'espressione `expr` al tipo `T`.

  ```
  static_cast<T>(expr)
  ```

- **Dynamic**
  Funge da supporto per la RTTI.
  Vengono anche utilizzati per effetuare conversioni all'interno di una gerarchia di classi legate da ereditarietá.

- **Const**
  Tipicamente utilizzato per rimuovere la qualificazione `const`.
  Si applica ad un riferimento (o puntatore) ad un oggetto qualificato `const` (ovvero non modificabile) per ottenere un riferimento (o puntatore) ad un oggetto non qualificato (e quindi modificabile).

  ```
  void promessa_marinaio(const int& ci) {

        int& i = const_cast<int&>(ci);
        ++i;
  }
  ```

  La funzione, in questo esempio, aveva promesso di non modificare l'argomento `ci`, ma tramite il cast elimina la qualificazione e modifica proprio l'argomento (non una copia).

- **Reinterpret**
  Si usa nei seguenti casi:

1. Da puntatore a intero (abbastanza grande da rappresentare il valore del puntatore).

2. Da intero a puntatore.

3. Da puntatore (o riferimento) a un altro tipo puntatore (o riferimento).

- **Funzionale**
  Usando le sintassi del tipo:

  ```
  T(expr)       T()
  ```

  Corrisponde semplicemente alla costruzione diretta di un oggetto di tipo T usando un costruttore (vuoto nel secondo caso).

  Nel caso di un tipo built-in la forma `T()` produce la cosiddetta zero-initialization.

- **Stile C**
  Hanno la seguente sintassi:

  ```
  (T) expr
  ```

  Il loro uso é considerato cattivo stile.
  Con questo tipo di cast é possibile simulare tutte le tipologie di cast viste finora ad eccezione dei `dynamic_cast`.

I Principi S.O.L.I.D.

Con questo acronimo si identificano i cinque principi della progettazione object oriented.

Lo scopo é quello di fornire delle linee guida verso uno sviluppo che renda il codice piú flessibile per facilitare manutenzione e estensione delle funzionalitá.

Si parla di principi e non regole poiché non sono strettamente obbligatori.
Sta al programmatore capire quando e come usarli per trarre i benefici elencati in precedenza.

- **SRP - Single Responsabilty Principle**
  É un principio di validitá generale.
  Dice che ogni porzione di codice che progettiamo deve avere in carico una responsabilitá (deve essere pensata per svolgere un compito preciso).
  Ad esempio, una classe che deve manipolare piú risorse (in exception safe) non dovrebbe prendersi la responsabilitá della gestione di acquisizione e rilascio di esse.

- **OCP - Open Closed Principle**
  Il principio "aperto-chiuso" dice, appunto, che il codice deve essere **aperto** alle estensioni e **chiuso** alle modifiche.

  In parole povere un software dovrebbe rendere facile l'aggiunta di nuove funzionalitá senza andare ad intaccare il codice giá scritto.

- **LSP - Liskov Substitution Principle**
  Ogni funzione che usa puntatori o riferimenti a classi base deve essere in grado di usare oggetti delle classi derivate senza saperlo.

  Le classi di un sottotipo `S` devono soddisfare le aspettative degli utenti che accedono ad esse tramite puntatori.
  Siccome `S` dichiara una relazione IS-A rispetto a `T`, gli oggetti di `S` devono comportarsi come gli oggetti di tipo `T` (fornendo ovviamente anche i medesimi metodi).

- **ISP - Interface Segregation Principle**
  Principio di separazione delle interfacce, dice che l'utente non deve essere forzato a dipendere da parti di un'interfaccia che non utilizza.

  Questo significa che il progettista deve cercare, per quanto possibile, di separare tutte le interfacce che possono essere usate separatamente.

  Questo porta diversi vantaggi come evitare la propagazione di errori oppure la possibilitá di modificare solo una piccola parte senza intaccare le altre.

- **DIP - Dependency Inversion Principle**
  I moduli di alto livello non devono dipendere da quelli di basso livello: entrambi devono dipendere da astrazioni.
  Le astrazioni non devono dipendere dai dettagli; sono i dettagli che dipendono dalle astrazioni.

  Questo principio opera quindi una classificazione sulle dipendenze tra

moduli software (classi, funzioni, ecc...) stabilendo quali sono le varie dipendenze tra esse.