26 - Classi dinamiche (lez. 04-12-2024)

Classi derivate e relazione "IS-A"

La relazione "IS-A" rappresenta uno dei principi fondamentali dell'ereditarietà in c++. Si verifica quando una classe derivata può essere considerata come una specializzazione della classe base.

```
class Base {
public:
    virtual void print() const { std::cout << "Base\n"; }
};

class Derived : public Base {
public:
    void print() const override { std::cout << "Derived\n"; }
};

int main() {
    Derived d;
    Base* base_ptr = &d; // Up-Cast implicito
    base_ptr->print();   // Output: Derived (se il metodo è virtuale)
}
```

In questo caso, la relazione "IS-A" si traduce nel fatto che `Derived` è una specializzazione di `Base`, quindi un oggetto `Derived` può essere usato ovunque sia richiesto un oggetto `Base`.

Derivazione non pubblica

Se la derivazione è **private** o **protected**, l'up-cast implicito non è consentito al di fuori del contesto della classe derivata o di una funzione `friend`. Questo serve a nascondere il fatto che una classe deriva da un'altra.

```
class Base {};

class Derived : private Base {}; // Derivazione privata

int main() {
    Derived d;
    Base* base_ptr = &d; // Errore: conversione non consentita
    return 0;
}
```

Metodi virtuali e classi dinamiche

I **metodi virtuali** consentono di implementare il **polimorfismo runtime**, una caratteristica fondamentale di C++. Grazie ai metodi virtuali, la risoluzione delle chiamate a una funzione membro non avviene a tempo di compilazione, ma a tempo di esecuzione, in base al tipo dinamico dell'oggetto.
Quando una funzione membro viene dichiarata `virtual` nella classe base, le sue ridefinizioni nelle

classi derivate vengono chiamate tramite un meccanismo di dispatch dinamico. Questo comportamento viene gestito utilizzando una struttura chiamata **vtable** (tabella virtuale).

```
class Base {
public:
    virtual void print() const {
        std::cout << "Base\n";
    }
};

class Derived : public Base {
public:
    void print() const override {
        std::cout << "Derived\n";
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->print(); // Output: Derived
    delete ptr;
}
```

In questo esempio:

1. la classe `Base` è una **classe dinamica** perché contiene un metodo virtuale;
2. quando `ptr->print()` viene chiamato, il compilatore non sa a quale versione del metodo `print` fare riferimento. Il **runtime** interroga la vtable per scoprire quale metodo chiamare in base al tipo dinamico dell'oggetto (`Derived`).
   In particolare:

```
Base* ptr = new Derived;
```

- il **tipo statico** di `ptr` è `Base*` perché `ptr` è dichiarato come puntatore a `Base` :
  - **tipo statico** := tipo che il compilatore conosce al momento della compilazione.
- il **tipo dinamico** di `*ptr` è `Derived` , perché `ptr` punta effettivamente a un oggetto della classe `Derived` :
  - **tipo dinamico** := tipo effettivo dell'oggetto a cui il puntatore (o riferimento) punta durante l'esecuzione.

Metodi virtuali puri e classi astratte

Metodi virtuali puri:

è un metodo dichiarato nella forma:

```
virtual void metodo() = 0;
```

Un metodo virtuale puro:

- non ha implementazione nella classe base;
- obbliga le classi derivate a fornire una propria implementazione.
  Le classi che contengono almeno un metodo virtuale puro sono dette **classi astratte** e non possono essere istanziate.

```
class Astratta {
public:
    virtual void metodo_puro() = 0; // Metodo virtuale puro
};

class Concreta : public Astratta {
public:
    void metodo_puro() override {
        std::cout << "Implementazione concreta\n";
    }
};

int main() {
    Astratta a; // Errore: la classe è astratta
    Concreta c;
    c.metodo_puro(); // Output: Implementazione concreta
}
```

Distruttori virtuali

Un distruttore virtuale è essenziale per garantire che la distruzione degli oggetti derivate avvenga correttamente quando si usa un puntatore alla classe base.
Se un distruttore non è virtuale, quando un oggetto derivato viene distrutto tramite un puntatore alla classe base, viene chiamato solo il distruttore della classe base, causando **memory leak**.
Esempio senza distruttore virtuale:

```
class Base {
public:
    ~Base() { std::cout << "Distruttore Base\n"; }
};

class Derived : public Base {
public:
    ~Derived() { std::cout << "Distruttore Derived\n"; }
};

int main() {
    Base* b = new Derived;
    delete b; // Output: Distruttore Base (ma non Distruttore Derived!)
}
```

Per evitare questo, il distruttore deve essere dichiarato `virtual` nella classe base:

```
class Base {
public:
```

```
    virtual ~Base() { std::cout << "Distruttore Base\n"; }
};
```

Risoluzione dell'overriding

Perché l'overriding funzioni correttamente, devono essere rispettate alcune condizioni:

1. **il metodo deve essere dichiarato virtuale** nella classe base;
2. **deve essere invocato tramite un puntatore o un riferimento** alla classe base;
3. **la classe derivata deve fornire un'implementazione** del metodo;
4. **non deve esserci qualificazione esplicita**: se si usa un qualificatore, si chiama esplicitamente il metodo della classe qualificata.
   Esempio di qualificazione esplicita:

```
class Base {
public:
    virtual void print() const {
        std::cout << "Base\n";
    }
};

class Derived : public Base {
public:
    void print() const override {
        std::cout << "Derived\n";
    }
};

int main() {
    Derived d;
    d.print();           // Output: Derived
    d.Base::print();     // Output: Base (qualificazione esplicita)
}
```