29 - Ereditarietà

Nel caso di polimorfismo dinamico, le classi astratte sono formate tipicamente da:

- metodi virtuali puri;
- il distruttore della classe dichiarato virtuale (ma non puro).
  In alcuni casi occorre complicare il progetto (es. usando l'ereditarietà multipla), nonostante si corra il rischio di fare errori.

Metodi che NON possono essere dichiarati virtuali**

- **Costruttori**:
  I costruttori **non possono essere dichiarati virtuali**. Questo perché, durante la costruzione di un oggetto, il meccanismo di polimorfismo dinamico non è ancora operativo (la vtable non è ancora completamente inizializzata). Per ottenere un comportamento "simile" al polimorfismo, si usa il design pattern *factory method* o l'ereditarietà virtuale per gestire la costruzione di oggetti attraverso puntatori o riferimenti a classi base.
- **Distruttori**:
  I distruttori **possono essere dichiarati virtuali** e spesso **devono esserlo** quando si lavora con polimorfismo dinamico. Questo garantisce che, quando si elimina un oggetto tramite un puntatore alla classe base, il distruttore della classe derivata venga chiamato correttamente, evitando problemi di *memory leak*.
  Esempio:

```
class Base {
public:
    virtual ~Base() {}
};

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

- **Funzioni membro (non statiche)**:
  Queste **possono essere dichiarate virtuali**, ed è proprio il contesto principale del polimorfismo dinamico. Solo le funzioni membro non statiche possono accedere al puntatore implicito `this`, che è essenziale per il comportamento polimorfico.
- **Funzioni membro statiche**:
  Le funzioni statiche **non possono essere dichiarate virtuali**. Questo perché non fanno parte dell'istanza dell'oggetto e non accedono al puntatore `this`. Di conseguenza, non possono partecipare alla risoluzione dinamica dei metodi.
- **Template di funzioni membro (non statiche)**:
  Le funzioni membro template **possono essere dichiarate virtuali**, ma raramente ha senso farlo. Ogni istanza del template genera un metodo separato per il tipo specifico, il che può rendere difficile mantenere un comportamento polimorfico coerente.

- **Funzioni membro di classi templatiche (non statiche)**:
  Le funzioni membro di classi templatiche **possono essere virtuali**, e ciò può essere utile quando si vuole implementare una gerarchia di classi basate su template.
  Esempio:

```
template <typename T>
class Base {
public:
    virtual void foo() = 0;
};

class Derived : public Base<int> {
public:
    void foo() override { std::cout << "Derived\n"; }
};
```

Come costruire una copia di un oggetto concreto dato un puntatore/riferimento alla classe base

Per creare una copia di un oggetto concreto dato un puntatore o un riferimento alla classe base, si può utilizzare il **metodo virtuale** `clone` . Questo approccio è comune nei pattern di progettazione, come il *Prototype pattern*.

```
class Base {
public:
    virtual ~Base() {}
    virtual Base* clone() const = 0;
};

class Derived : public Base {
public:
    Derived* clone() const override { return new Derived(*this); }
};
```

In questo modo, chiamando `clone()` su un oggetto tramite un puntatore alla base, si ottiene una copia corretta dell'oggetto derivato.

Invocare un metodo virtuale durante costruzione/distruzione

Invocare un metodo virtuale durante la costruzione o la distruzione di un oggetto può portare a comportamenti inaspettati. Durante queste fasi, il tipo dinamico dell'oggetto è considerato il tipo della classe in corso di costruzione o distruzione, non quello effettivo del derivato.

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

class Derived : public Base {
```

```
public:
    void foo() override { std::cout << "Derived::foo\n"; }
};

Derived d;
```

Output:

```
Base::foo
Derived::foo
```

Durante il costruttore, il metodo `foo` della classe `Base` viene chiamato perché il costruttore della classe derivata non è ancora stato eseguito. Durante il distruttore, succede il contrario.

Ereditarietà multipla con classi non astratte

L'ereditarietà multipla introduce complessità che devono essere gestite con attenzione.

Scope e ambiguità

Se due classi base contengono metodi con lo stesso nome, si può incorrere in ambiguità. È necessario specificare la classe di origine:

```
class A {
public:
    void foo() { std::cout << "A\n"; }
};

class B {
public:
    void foo() { std::cout << "B\n"; }
};

class Derived : public A, public B {};

Derived d;
d.foo(); // Errore: ambiguità
d.A::foo(); // Corretto
```

Classi base ripetute

Se una classe è ereditata più volte attraverso diverse catene di ereditarietà, si hanno copie multiple di quella classe base, causando ridondanza.

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ }; // D contiene due istanze di A
```

Classi base virtuali

Per evitare duplicazione di classi base, si utilizza l'ereditarietà virtuale:

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public B, public C { /* ... */ }; // Una sola istanza di A
```

Semantica speciale di inizializzazione

Con classi base virtuali, la classe derivata più specifica è responsabile dell'inizializzazione della classe base comune:

```
class A {
public:
    A(int x) { std::cout << "A: " << x << '\n'; }
};

class B : virtual public A {
public:
    B() : A(0) {} // Non inizializza A
};

class C : virtual public A {
public:
    C() : A(0) {} // Non inizializza A
};

class D : public B, public C {
public:
    D() : A(42), B(), C() {} // A viene inizializzata qui
};
```

Usi del polimorfismo dinamico nella libreria standard

Classi eccezione standard

Il polimorfismo dinamico è alla base del sistema di gestione delle eccezioni in C++. Tutte le eccezioni derivano da `std::exception`, che fornisce il metodo virtuale `what()`:

```
try {
    throw std::runtime_error("Errore!");
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Classi iostream

Le classi della gerarchia `iostream` utilizzano il polimorfismo dinamico per fornire interfacce comuni a flussi di input (`istream`), output (`ostream`), e combinati (`iostream`).

```
std::ostream& stream = std::cout;
stream << "Polimorfismo dinamico!\n";
```