

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Funzioni virtuali

Stefano Ghidoni



Agenda

- Ereditarietà e funzioni virtuali
- Funzioni virtuali e overriding
- Esempi di funzionamento: funzioni virtuali vs non virtuali
- Classi astratte (2)



Gerarchia di classi

- Con le gerarchie di classi utilizziamo tre meccanismi fondamentali
 - **Ereditarietà / derivazione:** una classe eredita funzioni e dati membro dalla classe base
 - **Funzioni virtuali:** possibilità di definire la stessa funzione nella classe base e in quella derivata
 - AKA **polimorfismo run-time** perché è a tempo di esecuzione che si determina quale funzione deve essere chiamata
 - **Incapsulamento:** membri private e protected per nascondere i dettagli implementativi
 - Semplifica la manutenzione



Funzioni virtuali

- Alcune funzioni sono dichiarate virtuali
 - Utili se sono reimplementate nelle classi derivate
 - Analizziamo meglio un caso: `draw_lines()`

```
class Shape {  
    // ...  
    virtual void draw_lines() const;  
    // ...  
};  
  
class Circle : public Shape {  
    // ...  
    void draw_lines() const;  
    // ...  
};
```



Overriding

- Una classe derivata che ridefinisce una funzione virtuale di una classe base effettua un **override**
 - Questo fa sì che la funzione nella classe derivata sfrutti l'interfaccia della classe base
- La funzione oggetto di override ha stesso nome, stessi tipi e stessa constness della funzione nella classe base
- L'overriding è un elemento chiave del polimorfismo run-time



Overriding e puntatori

- L'overriding ha un funzionamento interessante con i puntatori
- Consideriamo le classi B, D, DD:

```
class B {  
// ...  
};  
  
class D : public B {  
// ...  
};  
  
class DD : public D {  
// ...  
};
```



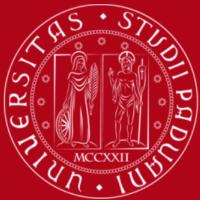
Overriding e puntatori

- L'overriding ha un funzionamento interessante con i puntatori
 - Un puntatore a B può puntare a un oggetto D o DD
 - Ricorda: D è un B, DD è un D e anche un B



Overriding e puntatori

- L'overriding ha un funzionamento interessante con i puntatori
 - Un puntatore a B può puntare a un oggetto D o DD
- Un `vector<B*>` può gestire una collezione di oggetti diversi (tutti derivati da B)
 - Chiamate alla stessa funzione virtuale a partire da una collezione di puntatori – per ciascun oggetto è chiamata la funzione appropriata



Virtual vs non virtual

- Vediamo ora alcuni esempi di funzionamento
 - Confrontiamo funzioni virtuali e non virtuali



Virtual vs non virtual

```
class B {  
public:  
    virtual void f() const { cout << "B::f "; }  
    void g() const { cout << "B::g "; }      // non virtuale  
};  
  
class D : public B {  
public:  
    void f() const { cout << "D::f "; }  
    void g() const { cout << "D::g "; }  
};  
  
class DD : public D {  
public:  
    void f() { cout << "DD::f "; }    // nessun  
                                         // override: non è const  
    void g() const { cout << "DD::g "; }  
};
```



Virtual vs non virtual

```
void call(const B& b)
{
    b.f();
    b.g();
}
```

Formalmente ho un oggetto di classe base,
quindi chiama la funzione della classe B.
Nella realtà non è detto che l'oggetto
passato sia di tipo B e non un suo derivato.

- Che oggetto può entrare in b?
 - B?
 - D?
 - DD?



Virtual vs non virtual

- D è un tipo di B, quindi può essere fornito come argomento
- DD è un tipo di D, quindi può essere fornito come argomento



Esempio di overriding

```
int main()
{
    B b;
    D d;
    DD dd;

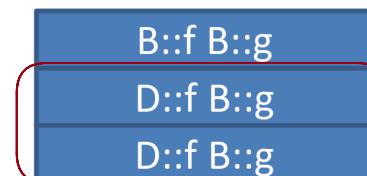
    call(b);
    call(d);
    call(dd);

    b.f();
    b.g();

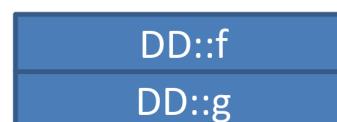
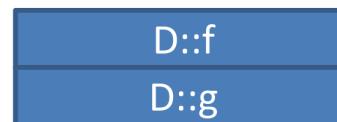
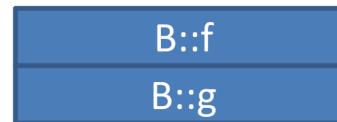
    d.f();
    d.g();

    dd.f();
    dd.g();
}
```

```
void call(const B& b)
{
    b.f();
    b.g();
}
```



La funzione call "si accorge" del reale tipo dell'oggetto, quindi anche dell'override effettuato sulla funzione



Non c'è più separazione tra oggetto reale e oggetto formale



Override esplicito

- In casi reali esistono funzioni espressamente progettate per l'override
- È possibile dichiarare esplicitamente questa intenzione
 - Keyword override
 - Genera un errore di compilazione se l'override non è implementato
 - Utile sia per chiarezza che per essere sicuri che l'override sia gestito correttamente



Override esplicito

```
class B {  
    virtual void f() const { cout << "B::f "; }  
    void g() const { cout << "B::g "; }      // non virtuale  
};  
  
class D : public B {  
    void f() const override { cout << "D::f "; }  
    void g() const override { cout << "D::g "; }  // errore  
};  
  
class DD : public D {  
    void f() override { cout << "DD::f "; } // errore  
    void g() const override { cout << "DD::g "; } // errore  
};
```

La funzione B::g non è virtuale ← Perché?

Manca "const" ← Perché?



Funzioni virtuali pure

- Funzioni virtuali pure: sono funzioni che esplicitamente non possono essere implementate nella classe base
- Rendono la classe virtuale pura
 - È impossibile istanziare oggetti di questa classe

```
class B {  
public:  
    virtual void f() = 0;      // è richiesto l'overriding  
    virtual void g() = 0;      // è richiesto l'overriding  
};  
  
B b;    // errore: B è virtuale pura
```



Funzioni virtuali pure

- Perché creare una funzione virtuale pura?
 - Per impedire di istanziare oggetti di una determinata classe
 - Può essere un effetto voluto
 - Per obbligare tutte le classi derivate a implementare una determinata funzione (l'overriding è obbligatorio per le funzioni virtuali pure)



Classi astratte

- Abbiamo visto (modulo precedente) che Shape è una classe astratta (o classe virtuale pura)
 - Per shape ciò era ottenuto rendendo i costruttori protected
 - Realizzato più correttamente con le funzioni virtuali pure



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Funzioni virtuali

Stefano Ghidoni



DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE