10.2 – Funzioni virtuali

Libro di testo:

- Capitoli 14.3, 14.4

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

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

1. **Ereditarietà / derivazione:**
   una classe eredita funzioni e dati membro dalla classe base

2. **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)

3. **Incapsulamento:**
   membri `private` e `protected` per nascondere i dettagli implementativi
   - Semplifica la manutenzione

Classe Shape (1)

```
class Shape {
public:
    void draw() const;
    void move(int dx, int dy);

    Point point(int i) const;
    int number_of_points() const;

    Shape(const Shape&) = delete;
    Shape& operator=(const Shape&) = delete;

    ~Shape() { }

    // ...
```

```
protected:
    Shape() { }
    Shape(initializer_list<Point> lst);

    void draw_lines() const;
    void add(Point p);
    void set_point(int i, Point p);

private:
    std::vector<Point> points;
    Color lcolor;
    Line_style ls;
    Color fcolor;
};
```

Classe Shape (2)

```
class Shape {
public:
    void draw() const;
    virtual void move(int dx, int dy);

    Point point(int i) const;
    int number_of_points() const;

    Shape(const Shape&) = delete;
    Shape& operator=(const Shape&) = delete;

    virtual ~Shape() { }

    // ...
```

```
protected:
    Shape() { }
    Shape(initializer_list<Point> lst);

    virtual void draw_lines() const;
    void add(Point p);
    void set_point(int i, Point p);

private:
    std::vector<Point> points;
    Color lcolor;
    Line_style ls;
    Color fcolor;
};
```

Funzioni virtuali

- Alcune funzioni sono dichiarate `virtual`
  - 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;
    // ...
};
```

Esempio di implementazione:
- Legge ogni punto in `vector<Point>`
- Disegna a schermo ogni punto

Esempio di implementazione:
- Le variabili `center` e `radius`
- Disegna a schermo la circonferenza

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 alla base del polimorfismo dinamico (run-time)**

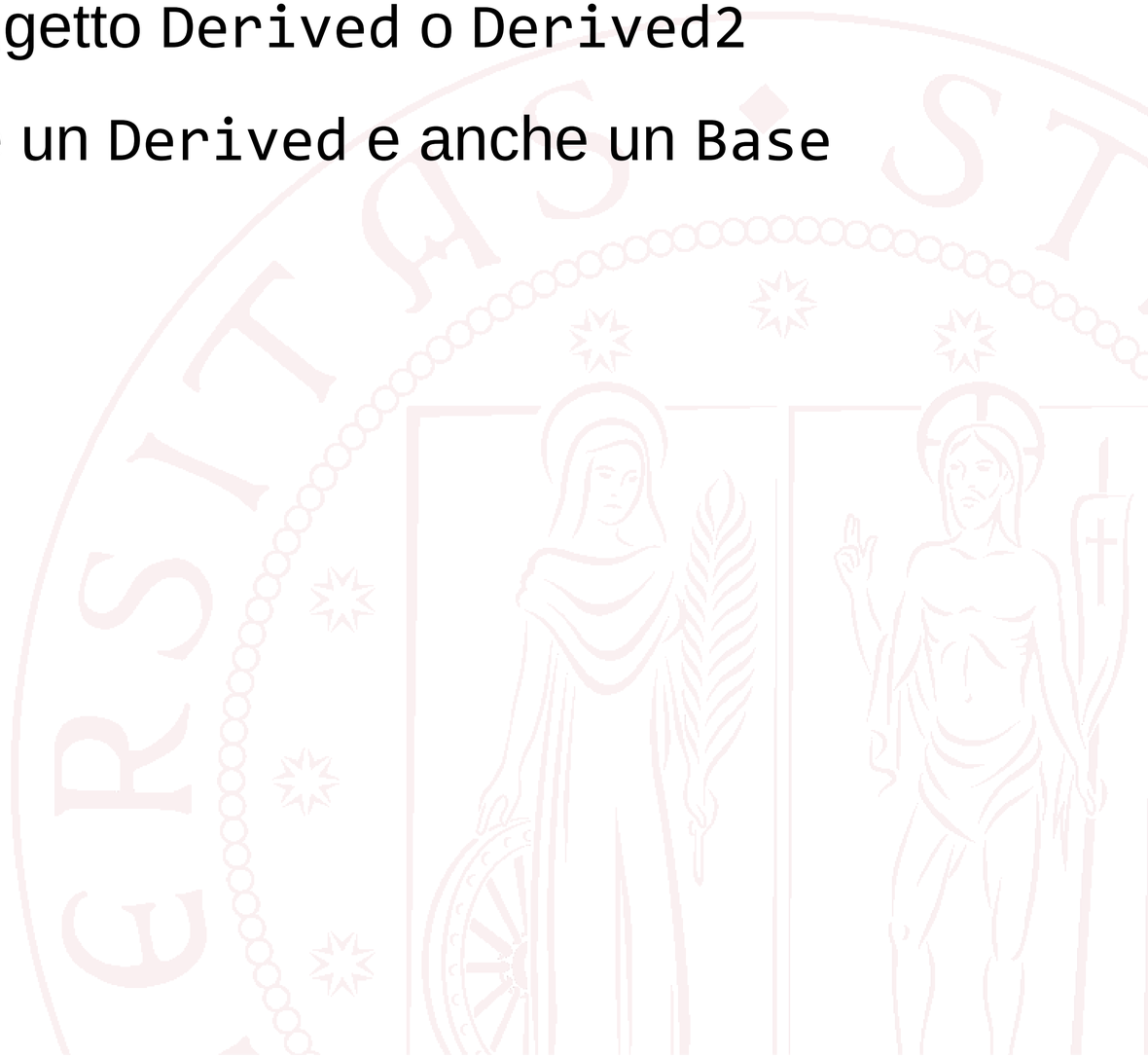
Overriding e puntatori

- L'overriding ha un funzionamento interessante con i puntatori
- Consideriamo le classi `Base, Derived, Derived2`:

```
class Base {
// ...
};

class Derived : public Base {
// ...
};

class Derived2 : public Derived {
// ...
};
```

Overriding e puntatori

- L'overriding ha un funzionamento interessante con i puntatori
  - Un puntatore a `Base` può puntare a un oggetto `Derived` o `Derived2`
  - Ricorda: `Derived` è un `Base`, `Derived2` è un `Derived` e anche un `Base`



Overriding e puntatori

- L'overriding ha un funzionamento interessante con i puntatori
  - Un puntatore a `Base` può puntare a un oggetto `Derived` o `Derived2`
- Un `vector<Base*>` può gestire una collezione di oggetti diversi

  (tutti derivati da `Base`)
  - Chiamate alla stessa funzione virtuale a partire da una collezione di puntatori – per ciascun oggetto è chiamata la funzione appropriata

```
std::vector<Base*> vd, vd2;

Derived D1, D2, D3;
Derived2 DD1, DD2, DD3
```

```
vd.push_back(&D1);
vd.push_back(&D2);
vd.push_back(&D3);

vd.push_back(&DD1);
vd.push_back(&DD2);
vd.push_back(&DD3);
```

Virtual vs. non virtual

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

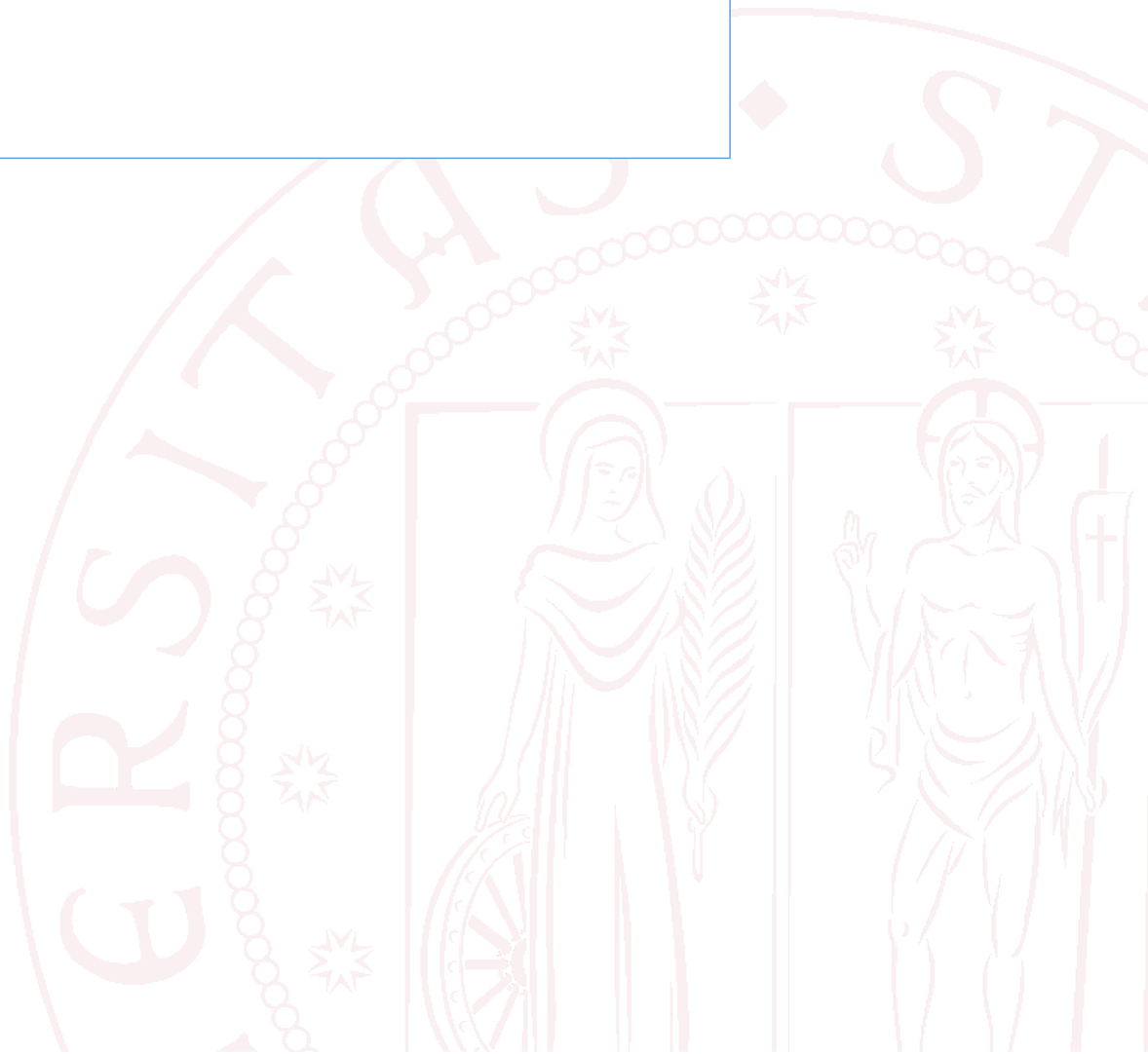
Virtual vs. non virtual

```
class Base {
   public:
      virtual void f() const { cout << "Base::f "; }
      void g() const { cout << "Base::g "; }       // non virtuale
};

class Derived : public Base {
   public:
      void f() const { cout << "Derived::f "; }
      void g() const { cout << "Derived::g "; }
};

class Derived2 : public Derived {
   public:
      void f() { cout << "Derived2::f "; }   // nessun override:
                                             // non è const
      void g() const { cout << "Derived2::g "; }
};
```

Virtual vs. non virtual

```
void call(const Base& base) {
    base.f();
    base.g();
}
```

- Che oggetto può entrare in base?
  - `Base`?
  - `Derived`?
  - `Derived2`?



Virtual vs. non virtual

- `Derived` è un tipo di `Base`, quindi può essere fornito come argomento
- `Derived2` è un tipo di `Derived`, quindi può essere fornito come argomento

Esempio di overriding

```
int main() {
    Base base;
    Derived derived;
    Derived2 derived2;

    call(base);
    call(derived);
    call(derived2);

    base.f();
    base.g();

    derived.f();
    derived.g();

    derived2.f();
    derived2.g();
}
```

```
void call(const Base& base)
{
    base.f();
    base.g();
}
```

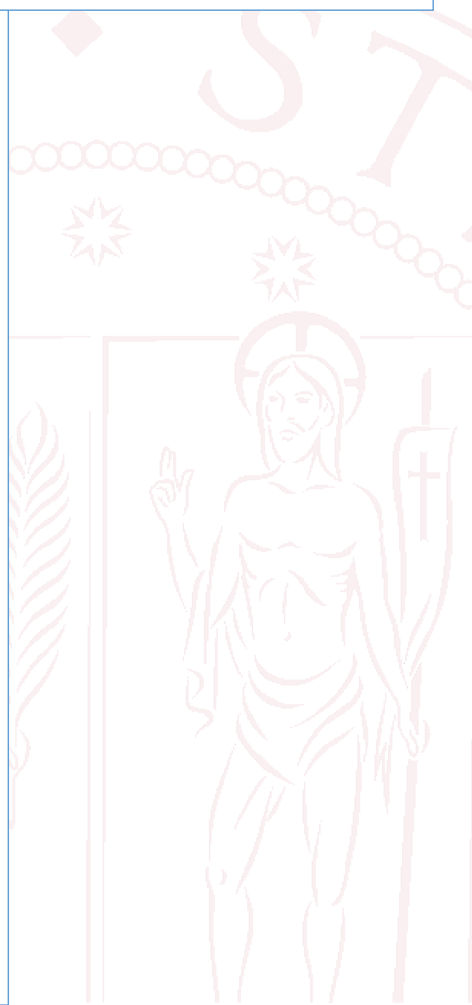
Qual è l'output?

Esempio di overriding

```
int main() {
    Base base;
    Derived derived;
    Derived2 derived2;

    call(base);
    call(derived);
    call(derived2);

    base.f();
    base.g();

    derived.f();
    derived.g();

    derived2.f();
    derived2.g();
}
```

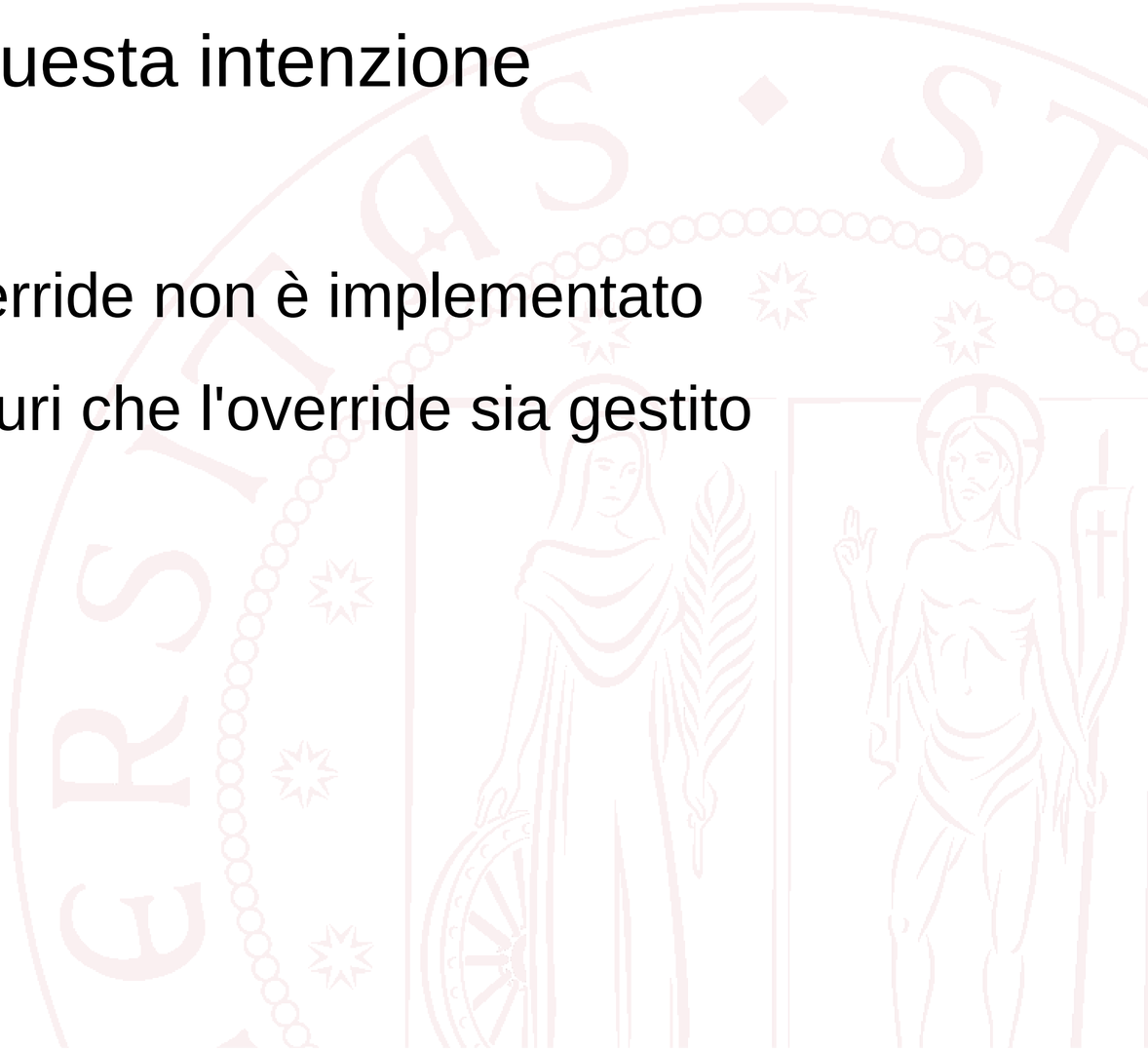
```
void call(const Base& base)
{
    base.f();
    base.g();
}
```

→	Base::f Base::g
→	Derived::f Base::g
→	Derived::f Base::g
→	Base::f
→	Base::g
→	Derived::f
→	Derived::g
→	Derived2::f
→	Derived2::g



Override esplicito

- In casi reali esistono funzioni espressamente progettate per l'override
- È possibile dichiarare esplicitamente questa intenzione
  - 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 Base {
    virtual void f() const { cout << "Base::f "; }
    void g() const { cout << "Base::g "; }   // non virtuale
};

class Derived : public Base {
    void f() const override { cout << "Derived::f "; }
    void g() const override { cout << "Derived::g "; }   // errore
};

class Derived2 : public Derived {
    void f() override { cout << "Derived2::f "; }         // errore
    void g() const override { cout << "Derived2::g "; } // errore

};
```

Perché?

Perché?

- La caratteristica `virtual` viene ereditata da tutte le classi derivate

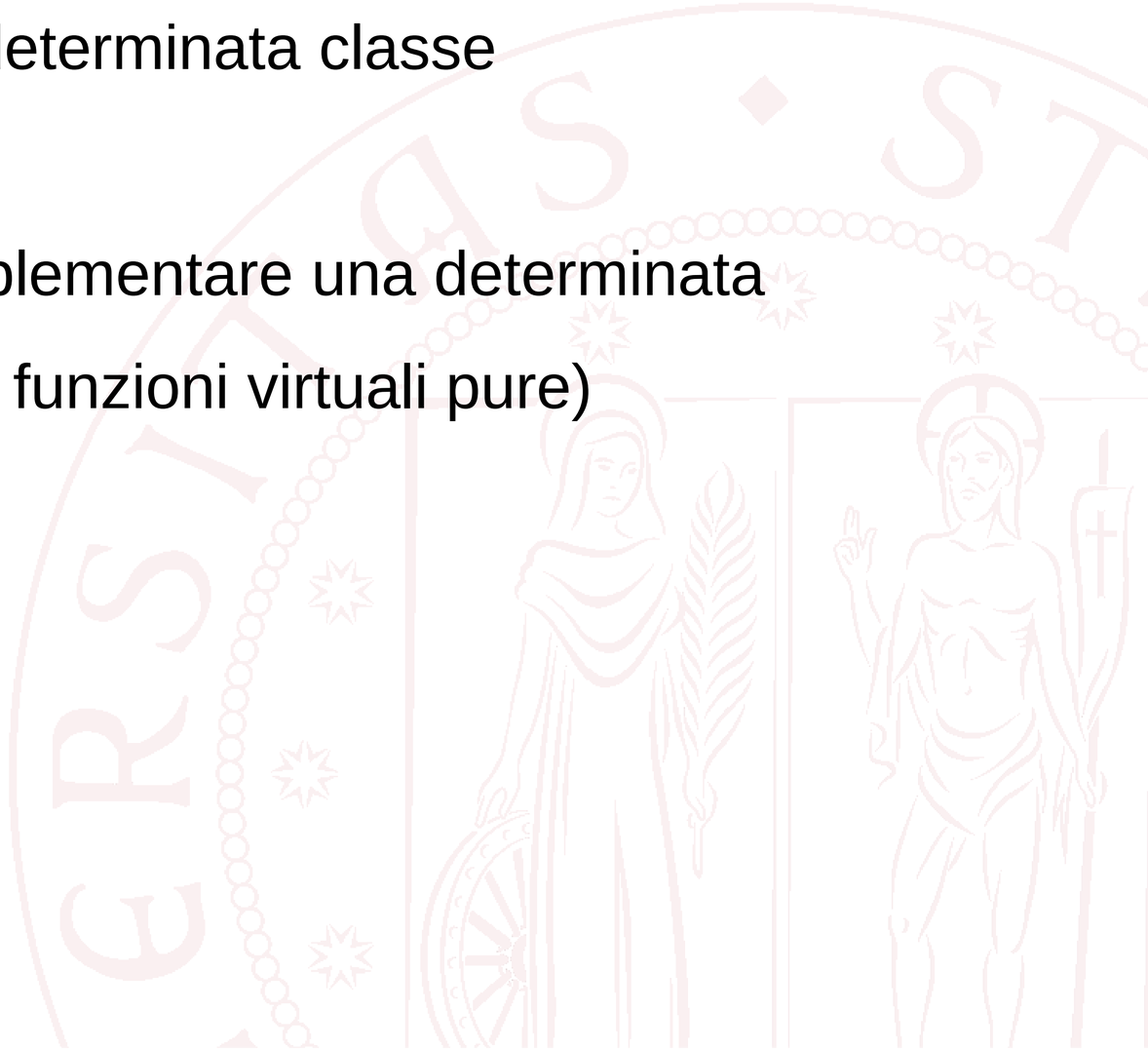
Funzioni virtuali pure

- Sono funzioni che esplicitamente non possono essere implementate nella classe base
- Rendono la classe **virtuale pura**
  - È impossibile istanziare oggetti di questa classe

```
class Base {
   public:
      virtual void f() = 0;   // è richiesto l'overriding
      virtual void g() = 0;   // è richiesto l'overriding
};

Base base;  // errore: Base è virtuale pura
```

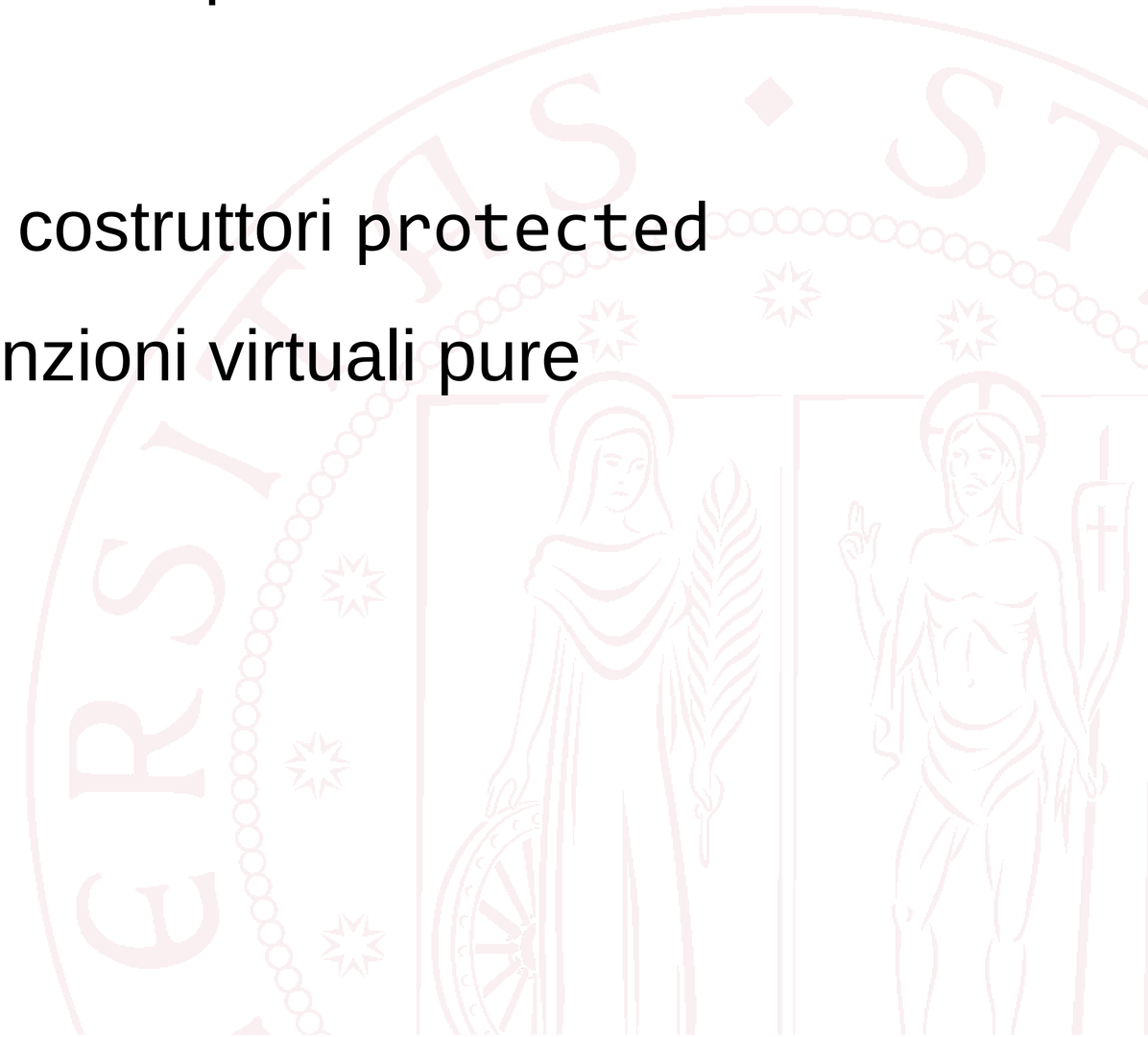
Funzioni virtuali pure

- 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



Classe Shape (3)

```
class Shape {
public:
    Shape() { }
    Shape(initializer_list<Point> lst);

    void draw() const;
    virtual void move(int dx, int dy) = 0;

    Point point(int i) const;
    int number_of_points() const;

    Shape(const Shape&) = delete;
    Shape& operator=(const Shape&) = delete;

    virtual ~Shape() { }

    // ...
```

```
protected:

    virtual void draw_lines() const = 0;
    void add(Point p);
    void set_point(int i, Point p);

private:
    vector<Point> points;
    Color lcolor;
    Line_style ls;
    Color fcolor;
};
```

Recap

- Overriding
- Funzioni virtuali e non virtuali
  - Meccanismo
  - Esempi di funzionamento
- Override esplicito
- Funzioni virtuali pure
- Classi astratte (2)