10.1 – Ereditarietà

Libro di testo:

- Capitoli 14.1, 14.2, 14.2.4

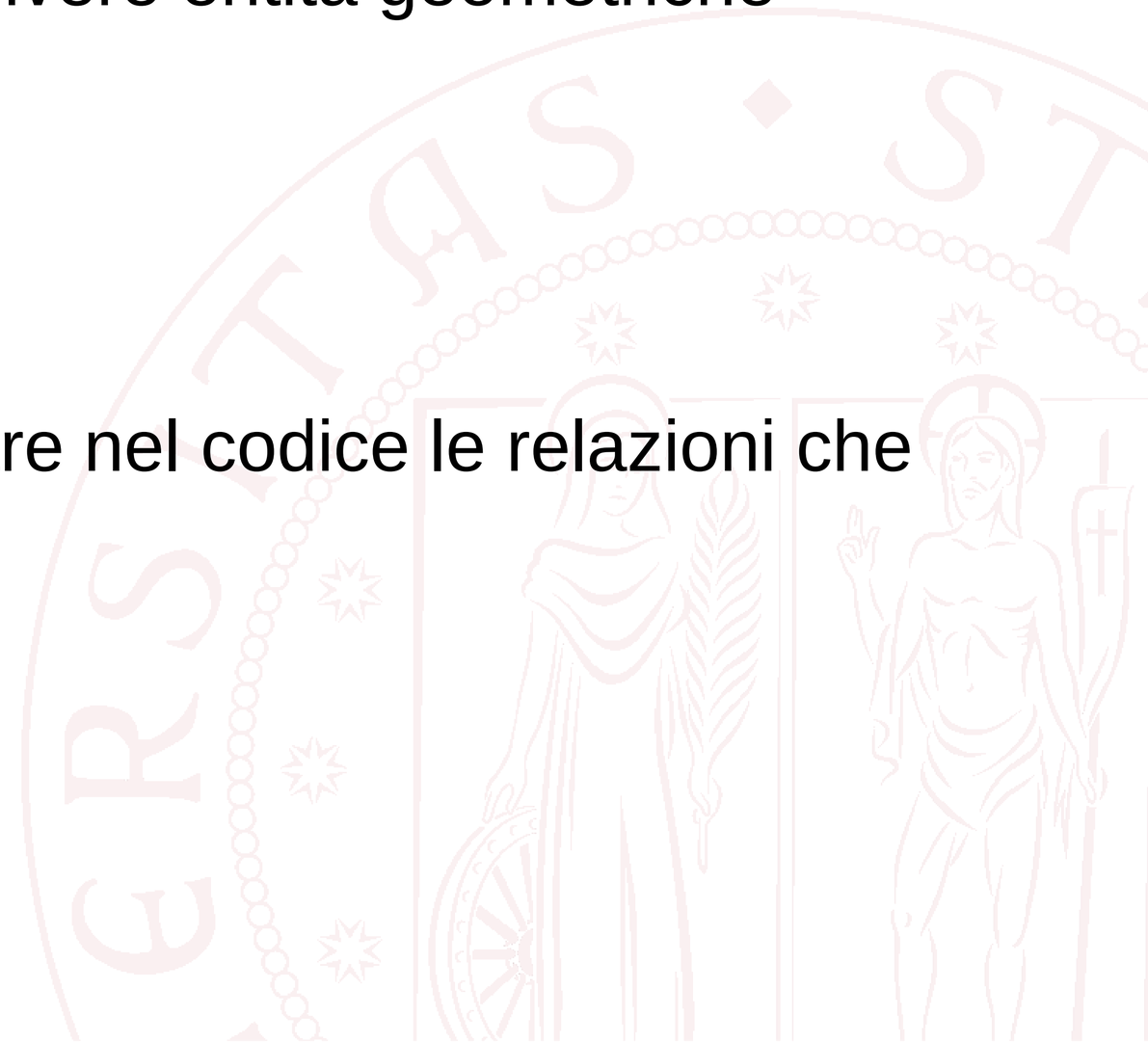
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Ereditarietà / gerarchie di classi
- Un esempio
- Classi astratte (1)
- Slicing

Descrivere entità geometriche

- Vogliamo creare una libreria per descrivere entità geometriche
  - Punti
  - Forme
  - ...
- Che classi creare? Come rappresentare nel codice le relazioni che esistono tra le varie forme?



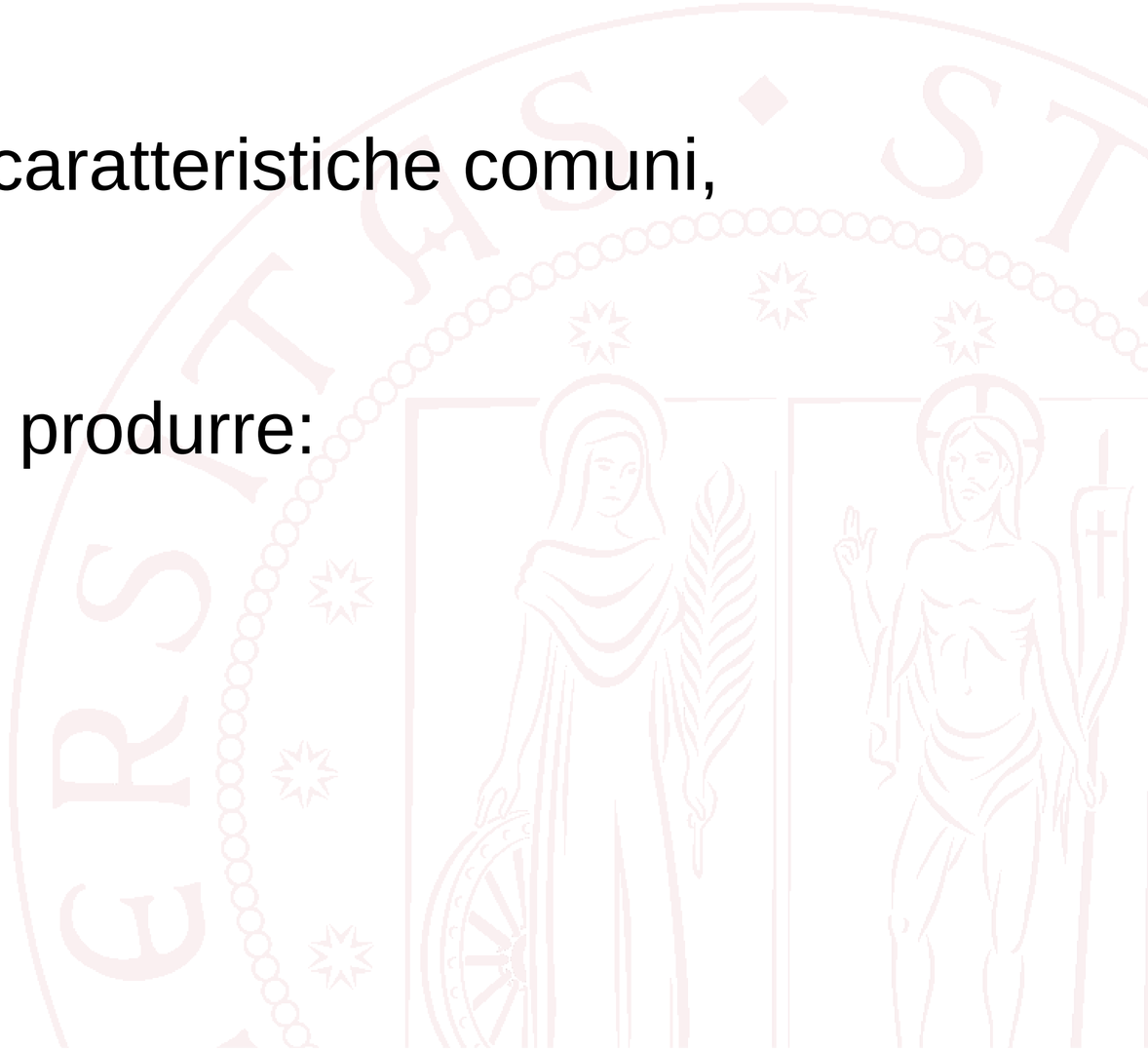
Entità da rappresentare

- Come definire una forma geometrica?
- Una forma possiede:
  - Alcune caratteristiche comuni
  - Descritta da un insieme di punti
  - Funzionalità disegno
- Alcune caratteristiche che dipendono dalla specifica forma
  - Non è possibile specificarle finché parliamo di forma in senso generale

Criteri di progettazione

- È possibile raggruppare le caratteristiche comuni
  - Classe base
- Le classi più specifiche **ereditano** tali caratteristiche comuni,

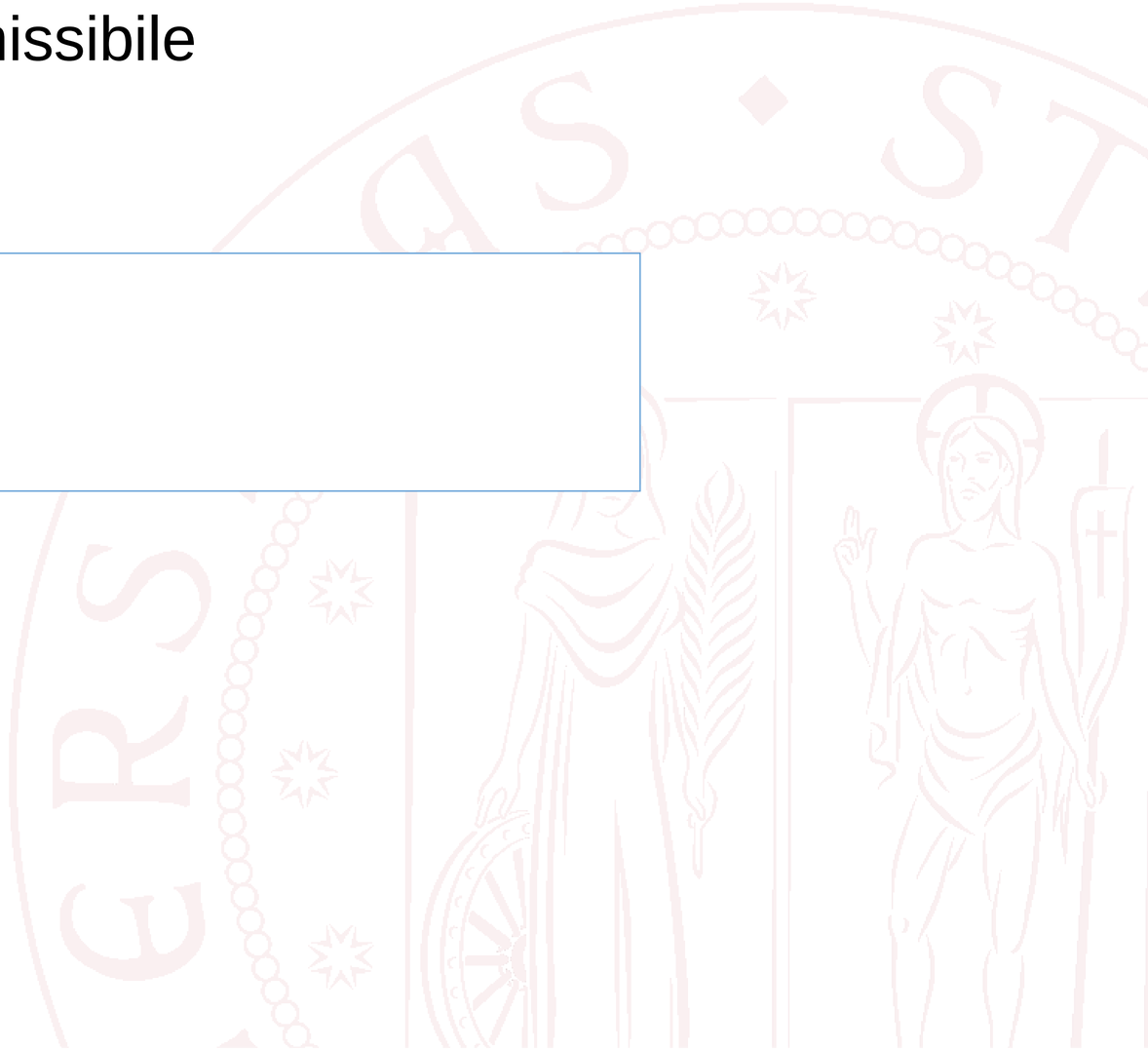
  e le estendono
- Una scelta progettuale usata spesso è produrre:
  - Molte classi
  - Con poche operazioni



Point

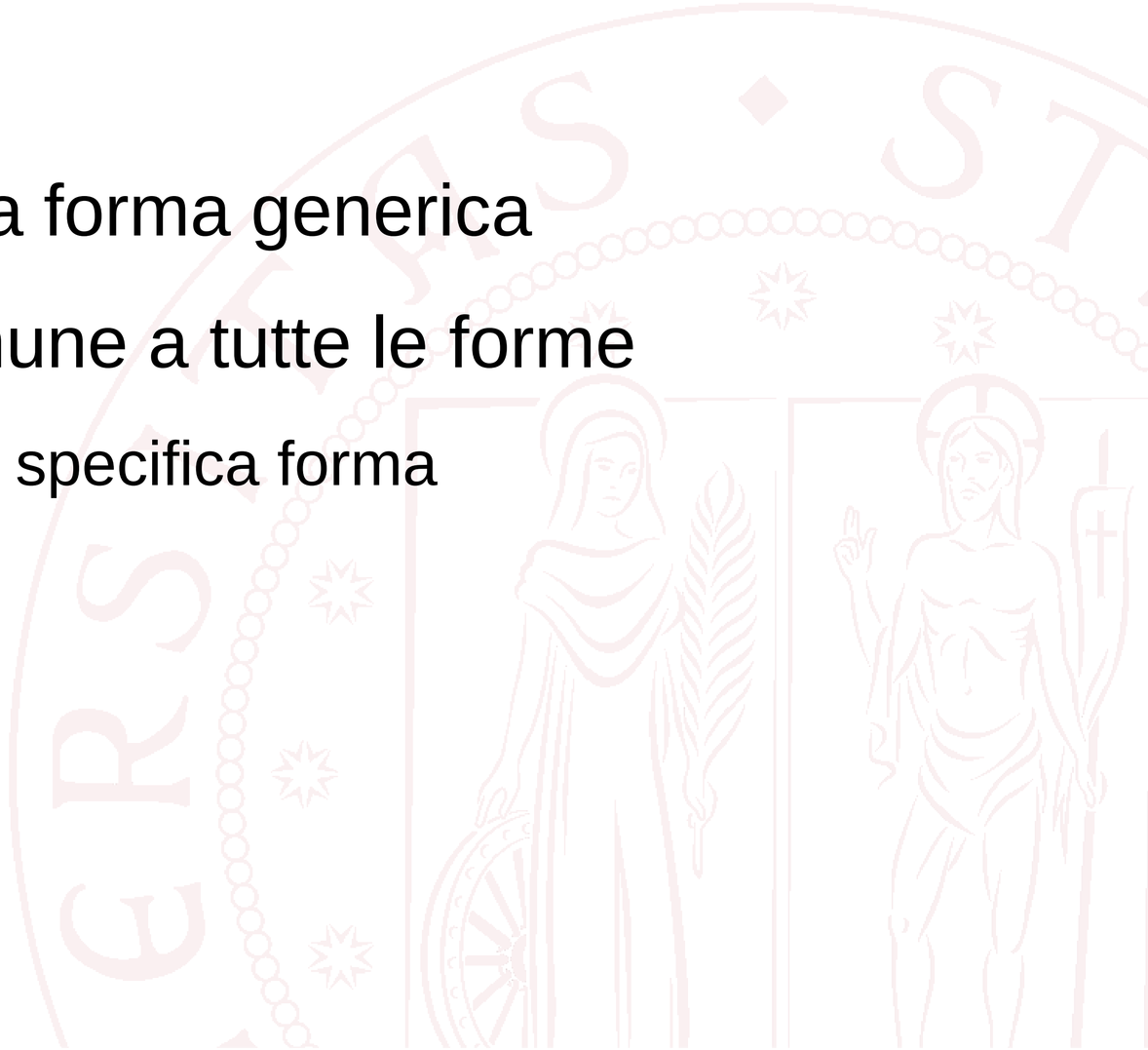
- L'entità base è il punto
  - Definito come `struct`: ogni valore è ammissibile
  - Raggruppa le due coordinate

```
struct Point {
    int x, y;
};
```



Shape

- `Shape` rappresenta una forma *generica*
  - Costruita a partire da `Point`
- Molte operazioni sono definibili per una forma generica
- `Shape` implementa **un'interfaccia** comune a tutte le forme
  - L'effettiva implementazione dipende dalla specifica forma



Classe Shape

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

        Point point(int i) const;          // accesso ai punti
        int number_of_points() const;

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

        virtual ~Shape() { }

        // ...
        // continua..
```

ATTENZIONE: NON È LA VERSIONE DEFINITIVA

Classe Shape

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

    void draw_lines() const;              // disegno
    void add(Point p);                    // aggiunge un punto
    void set_point(int i, Point p);       // points[i] = p;

private:
    std::vector<Point> points; // non usato da tutte le Shape
    Color lcolor;              // dalla libreria grafica
    Line_style ls;             // dalla libreria grafica
    Color fcolor;              // dalla libreria grafica
};
```

- Nota: line color vs fill color

ATTENZIONE: NON È LA VERSIONE DEFINITIVA

Classi astratte

- `Shape` è una classe astratta
  - Costruttore di default e con inizializzazione sono `protected`

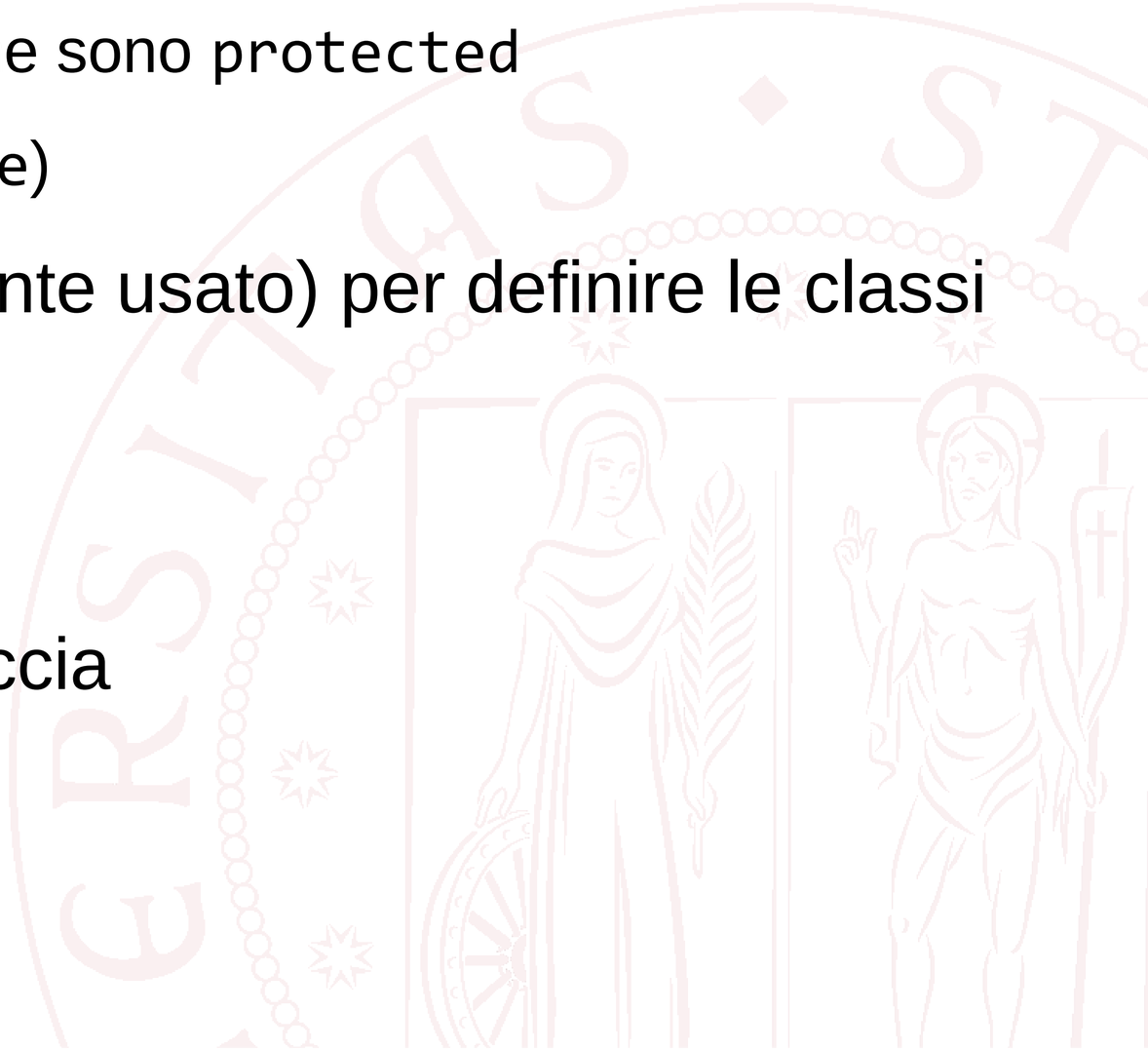
    (in questo caso, come se fossero `private`)

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

Classi astratte

- `Shape` è una classe astratta
  - Costruttore di default e con inizializzazione sono `protected`

    (in questo caso, come se fossero `private`)
- Esiste un altro modo (più frequentemente usato) per definire le classi astratte
  - **Funzioni virtuali pure** (vedi oltre)
- Classe astratta: implementa un'interfaccia



Ereditarietà

- È possibile definire classi che ereditano dalla classe `Shape`

```
class Circle : public Shape {
    // ...
};
```

- La classe `Circle` è derivata di `Shape`
  - Cosa eredita?

Ereditarietà

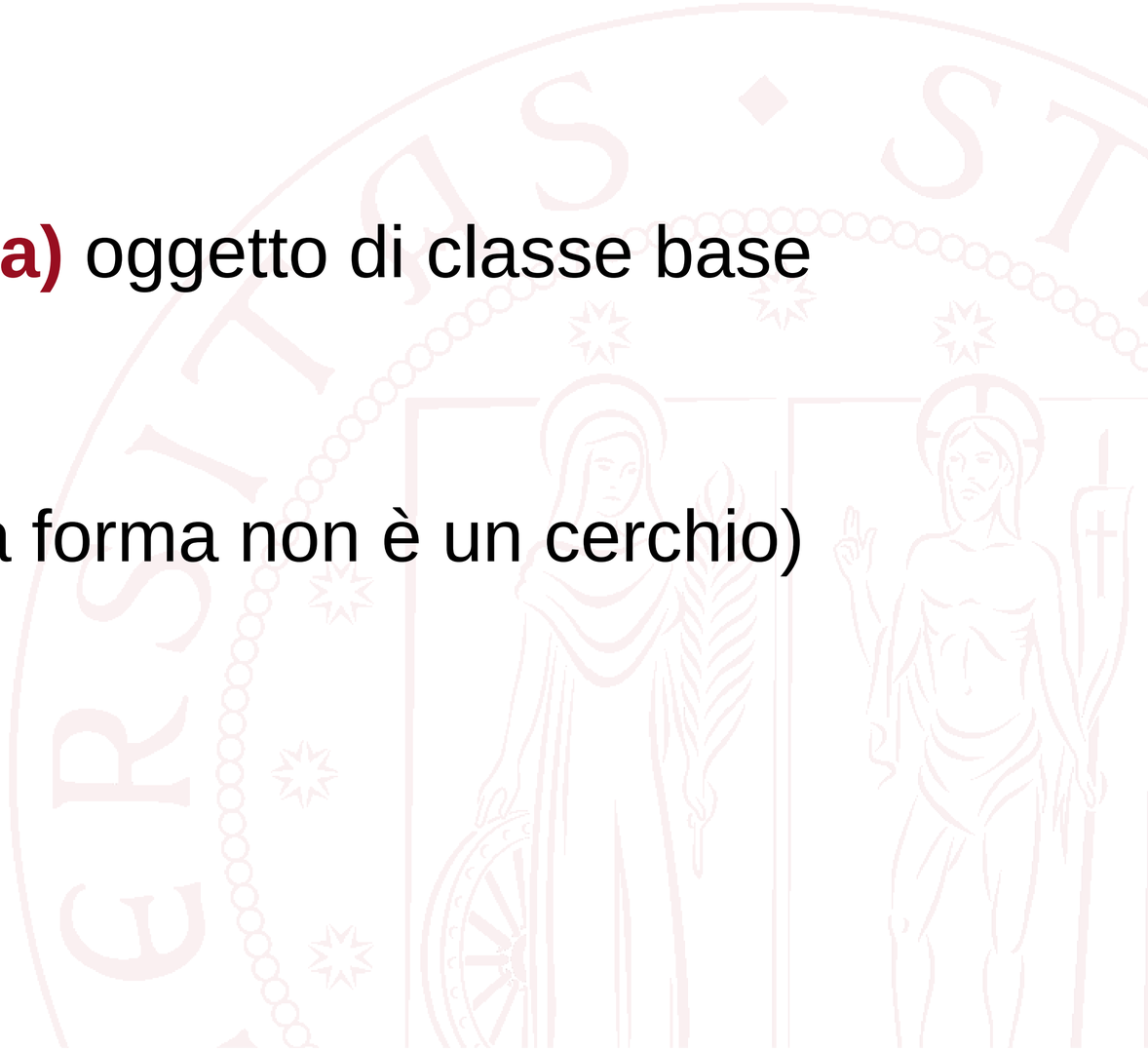
- È possibile definire classi che ereditano dalla classe `Shape`

```
class Circle : public Shape {
    // ...
};
```

- Ereditarietà **`public`**
  - La classe derivata può accedere ai membri `public` e `protected` della classe base
- Altri tipi di ereditarietà:
  - `private` (raramente usata)

“is a”

- Qual è il rapporto tra una classe base e la sua derivata?
  - Relazione "is a"
- Un oggetto di classe derivata **è un (is a)** oggetto di classe base
  - Es: un cerchio **è una** forma geometrica
- Questa relazione non è biunivoca (una forma non è un cerchio)



“has a”

- Un’altra possibile relazione tra classi è **"has a"**
- **Qualche esempio?**

“has a”

- Un’altra possibile relazione tra classi è **"has a"**
- **Qualche esempio?**
- **"has a"** definisce la composizione tra classi

  (un oggetto di una classe contenuto in un'altra classe)
  - Es: un'automobile **ha** un volante
  - Es: una forma **ha** dei punti

```
class Shape {
    public:
        //...
    private:
        std::vector<Point> points;
};
```

Slicing

- La relazione **is a** rende possibili alcune operazioni problematiche
  - Es: potrei inserire un oggetto `Circle` in un `std::vector` di `Shape`
- **Quali potenziali problemi?**

```
void my_fct(const Circle& c)
{
    vector<Shape> v;
    v.push_back(c);
}
```

Slicing

```
void my_fct(const Circle& c)
{
    vector<Shape> v;
    v.push_back(c);
}
```

- Se `Circle` è derivata di `Shape`, probabilmente ha dati membro aggiuntivi
  - Per esempio: `Circle` ha un raggio

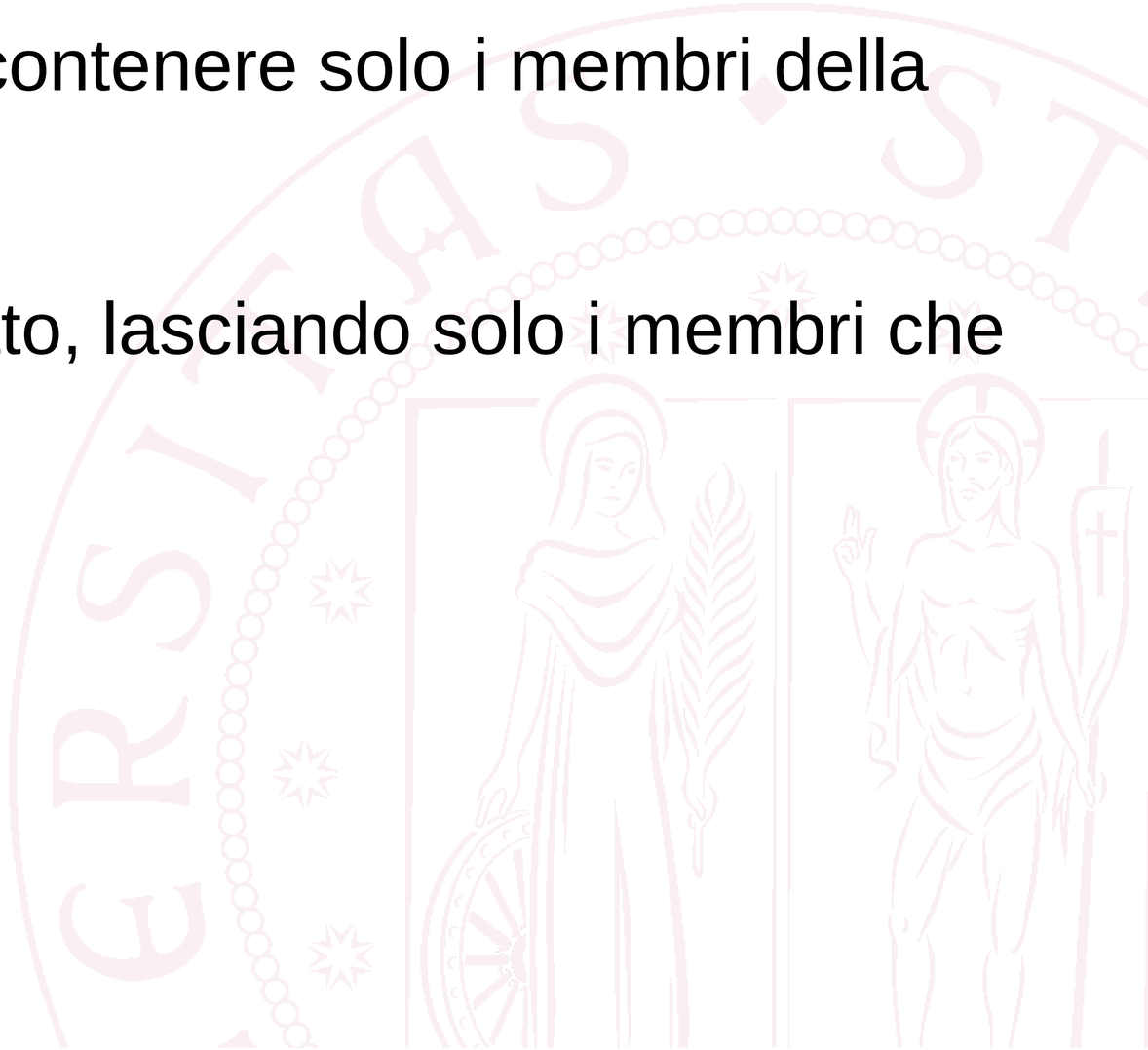
Slicing

```
void my_fct(const Circle& c)
{
    vector<Shape> v;
    v.push_back(c);
}
```

- Se `Circle` è derivata di `Shape`, probabilmente ha dati membro aggiuntivi
  - Per esempio: `Circle` ha un raggio
- Utilizzare il costruttore di copia (tramite il `push_back`) causerebbe un fenomeno di **slicing**

Slicing

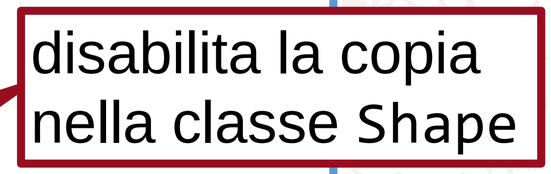
- Lo **slicing** si verifica quando tentiamo di inserire una classe derivata (es., `Circle`) in uno slot che riesce a contenere solo i membri della classe base
- Il compilatore risolve *affettando* l'oggetto, lasciando solo i membri che trovano posto



Slicing

- Costruttore di copia e `operator=` **devono** essere disabilitati tramite la keyword **`delete`**
  - Evita la creazione di costruttore di copia e `operator=` di default
  - Altrimenti, problemi di **slicing**

```
// Reminder
class Shape {
   public:
   // ...
        Shape(const Shape&) = delete;
        Shape& operator=(const Shape&) = delete;
   // ...
};
```



Slicing

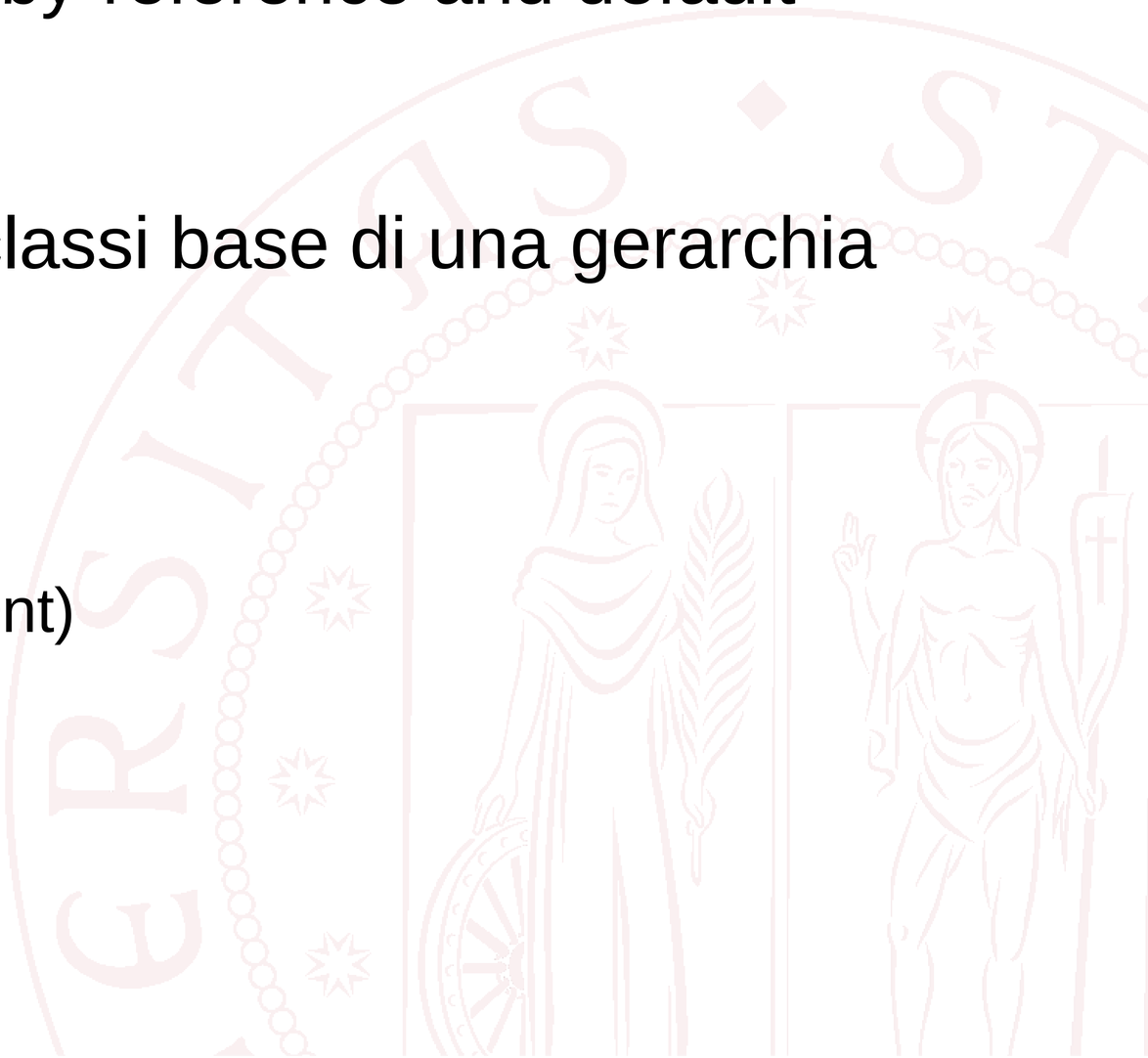
- Costruttore di copia e `operator=` **devono** essere disabilitati tramite la keyword **`delete`**
  - Evita la creazione di costruttore di copia e `operator=` di default
  - Ora la seconda riga causa **errore di compilazione**

```
void my_fct(const Circle& c)
{
    vector<Shape> v;
    v.push_back(c);          // errore: costruttore di copia
                             // disabilitato
}
```

(vedremo in seguito come gestire questo caso)

Disabilitare la copia

- "Basically, class hierarchies plus pass-by-reference and default copying do not mix" (BS)
- Le classi create per funzionare come classi base di una gerarchia richiedono di **disabilitare**:
  - Costruttore di copia (copy constructor)
  - Assegnamento di copia (copy assignment)



Recap

- Ereditarietà: definizione
- Classi astratte (1)
- Problema dello slicing
  - Gestione opportuna di costruttore e assegnamento di copia