13.3 – Smart pointer

Capitoli libro:

- 19.5.4

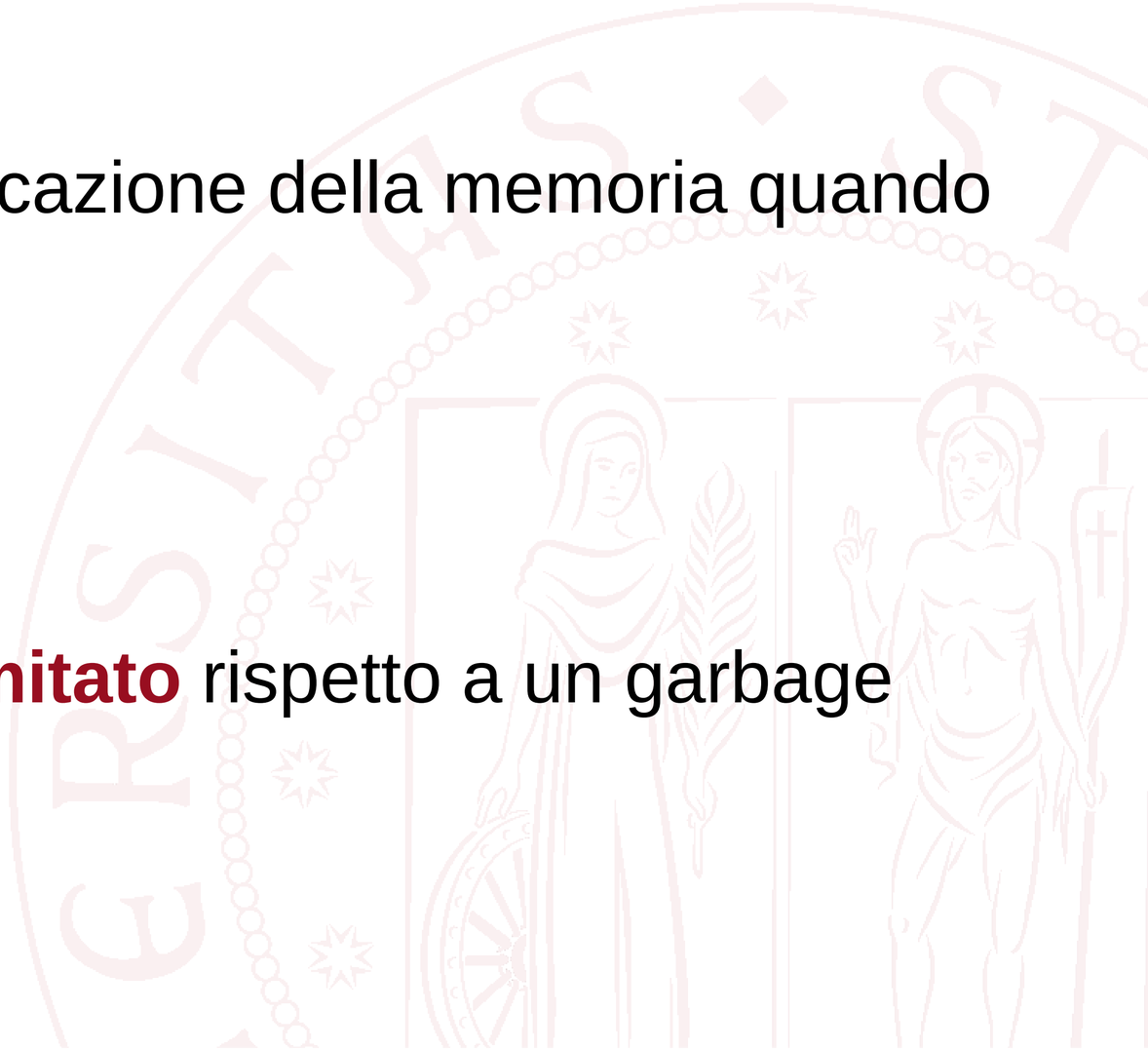
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Smart pointer: concetto
- I due tipi principali di smart pointer
- Esempi

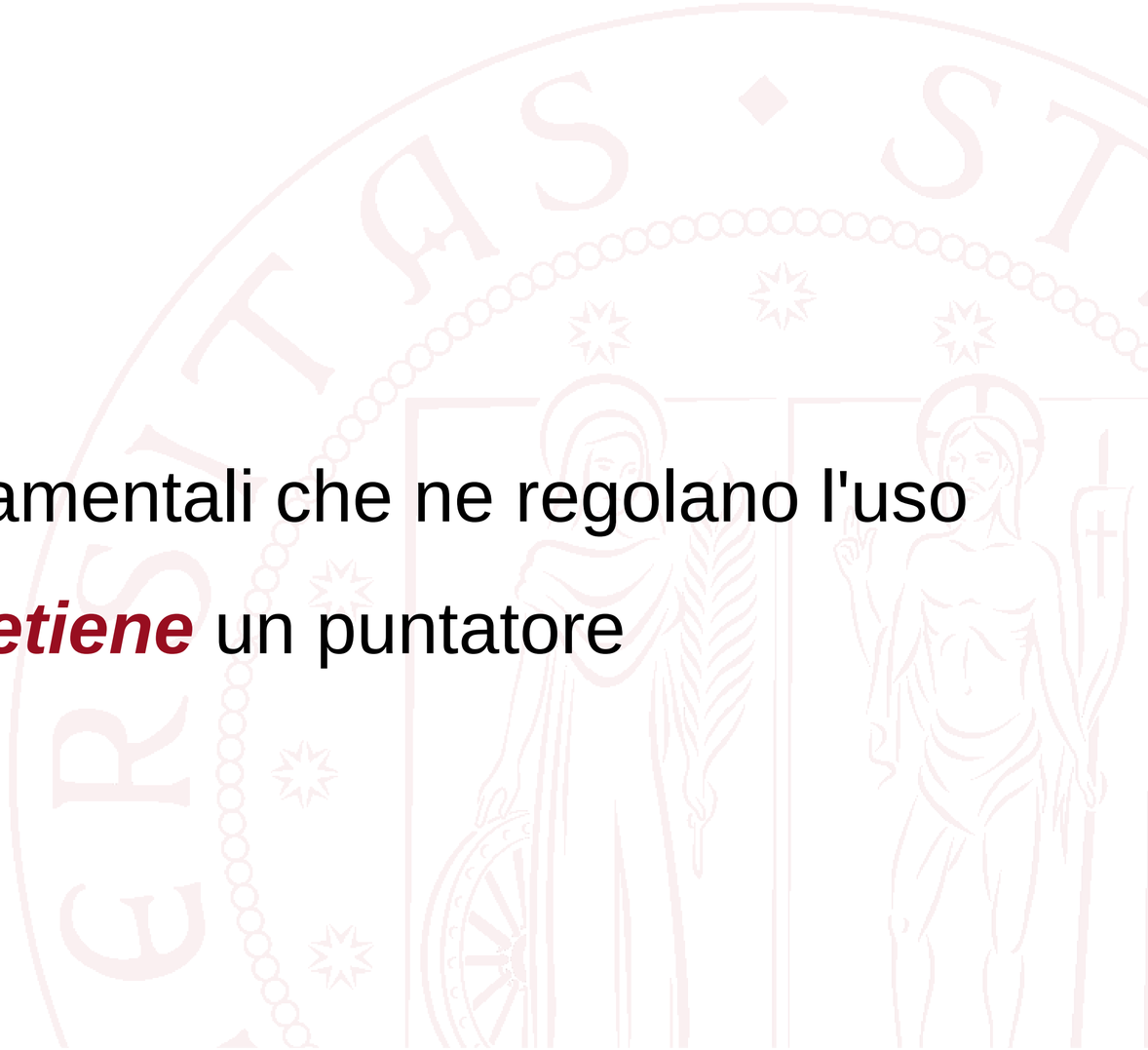
Smart pointer

- Puntatori intelligenti
- Classi template definite in `<memory>`
- Gestiscono automaticamente la deallocazione della memoria quando escono dallo scope
  - Eliminano memory leak
  - Eliminano dangling pointer
- Un overhead è presente, ma **molto limitato** rispetto a un garbage collector



Smart pointer

- Esistono vari tipi di smart pointer
  - `std::unique_ptr`
  - `std::shared_ptr`
  - ~~`std::auto_ptr` (deprecated)~~
- Diversi per alcune caratteristiche fondamentali che ne regolano l'uso
- Si è soliti dire che uno smart pointer ***detiene*** un puntatore



`std::unique_ptr`

- `std::unique_ptr` è uno smart pointer che:
  - Dealloca la memoria uscendo dallo scope
  - Permette i `move`
  - **Non permette le copie**
    - Come si ottiene questo effetto?
    - Costruttore e assegnamento di copia disabilitati
- **Non ha sostanziali overhead** rispetto a un puntatore

`std::unique_ptr`

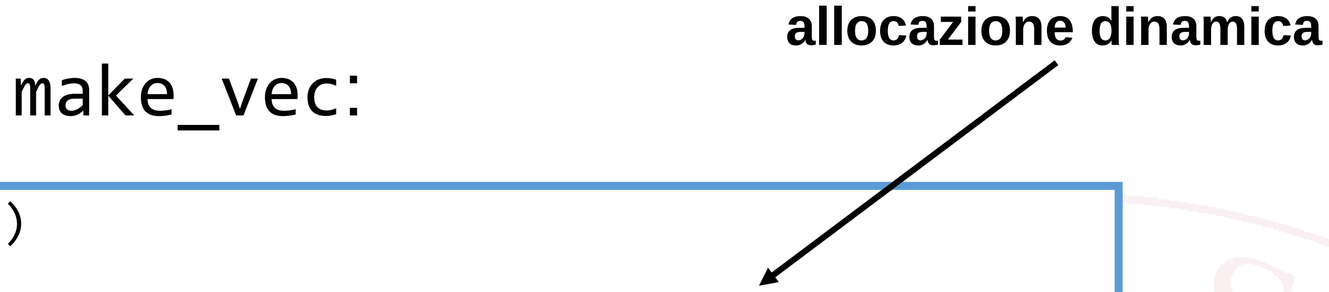
- Inizializzazione
  - Nel costruttore, oppure:
  - Funzione `reset()`
- Re-inizializzazione: funzione `reset()`
  - Fa sì che `std::unique_ptr` detenga un nuovo puntatore
  - Se `std::unique_ptr` deteneva un puntatore prima della chiamata, l'oggetto puntato è distrutto
  - Senza argomenti: semplice deallocazione

Uso e rilascio

- **Accesso**
  - È possibile usare gli operatori `*` e `->` come se fosse un puntatore
- **Liberazione automatica della memoria**
  - Quando `std::unique_ptr` è distrutto, cancella l'oggetto puntato!
- **Rilascio del puntatore**
  - `release()` *estrae* il puntatore dallo `std::unique_ptr`, il quale è resettato a `nullptr`

std::unique_ptr

- Esempio di utilizzo – riscriviamo `make_vec`:




```
std::vector<int>* make_vec()
{
    std::unique_ptr<std::vector<int>> p { new std::vector<int> };
    // ... riempimento del vettore con i dati – può
    // lanciare un'eccezione!
    return p.release();
}
```

- Inizializzazione nel costruttore
- Release del puntatore:
  - a fine utilizzo
  - Quando è lanciata un'eccezione
  - Quindi in `make_vec` **non è necessario fare il `catch`**!

std::unique_ptr

- Esempio di utilizzo – riscriviamo `make_vec`:

```
std::vector<int>* make_vec()
{
    std::unique_ptr<std::vector<int>> p { new std::vector<int> };
    // ... riempimento del vettore con i dati – può
    // lanciare un'eccezione!
    return p.release();
}
```

- `release()` *estrae* il puntatore dallo `std::unique_ptr`, il quale è resettato a `nullptr`
  - Quindi la successiva distruzione di `p` non cancella nulla
  - Si perde la deallocazione automatica

`std::unique_ptr`

Nota: allocare dinamicamente uno `std::vector` è solitamente

dannoso e sbagliato

Usato come esempio perché il riempimento può causare un'eccezione

Copia e spostamento

- Uno `std::unique_ptr`:
  - Non può essere copiato
  - Può essere spostato!
- È possibile ritornare uno `std::unique_ptr`
  - Sarà usato il move constructor

```
std::unique_ptr<int> AllocateAndReturn()
{
    std::unique_ptr<int> ptr_to_return(new int(42));

    return ptr_to_return;
}
```

Copia e spostamento

La copia è invece disabilitata

```
int main()
{
    std::unique_ptr<int> moved_ptr = AllocateAndReturn();
    // Errore di compilazione!
    std::unique_ptr<int> copied_ptr = moved_ptr;

     std::cout << *moved_ptr << std::endl;

    return 0;
}
```

Copia e spostamento

- Possiamo usare `std::unique_ptr` anche per risolvere il problema del `delete` nel chiamante
- Basta ritornare lo `std::unique_ptr`!

```
std::unique_ptr<std::vector<int>> make_vec()
{
    std::unique_ptr<std::vector<int>> p { new std::vector<int> };
    // ... riempimento del vettore con i dati – può
    // ritornare un'eccezione!
    return p;
}
```

Restrizioni

- `std::unique_ptr` ha una restrizione:

  due `std::unique_ptr` non possono puntare allo stesso oggetto

- Come detto, la copia tra `std::unique_ptr` è disabilitata

```
void no_good()
{
    std::unique_ptr<X> p { new X };
    std::unique_ptr<X> q { p };  // errore
}                                // qui sia p che q eliminerebbero l'oggetto
```

Restrizioni

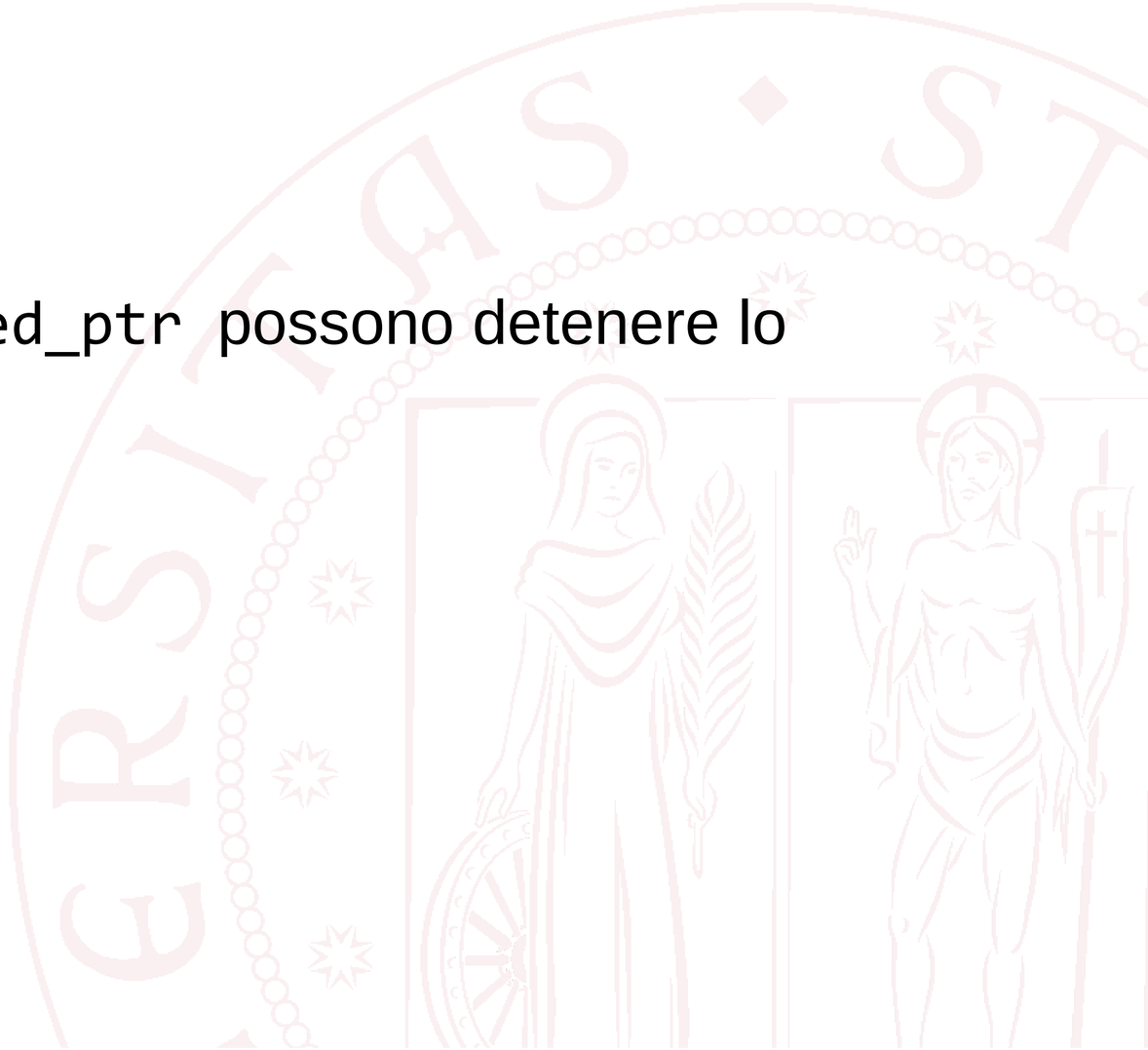
- Inizializzare due `std::unique_ptr` con lo stesso puntatore è un errore
  - Doppio `delete`
  - Non rilevabile dal compilatore

```
void no_good()
{
    int* p = new int;
    std::unique_ptr<int> sp {p};
    std::unique_ptr<int> sp2 {p};    // errore logico, ma compila
}                                    // qui sia sp che sp2 eliminano l'oggetto
```

- L'esistenza del puntatore da incapsulare è un punto debole
  - Risolvibile con `make_unique` – laboratorio

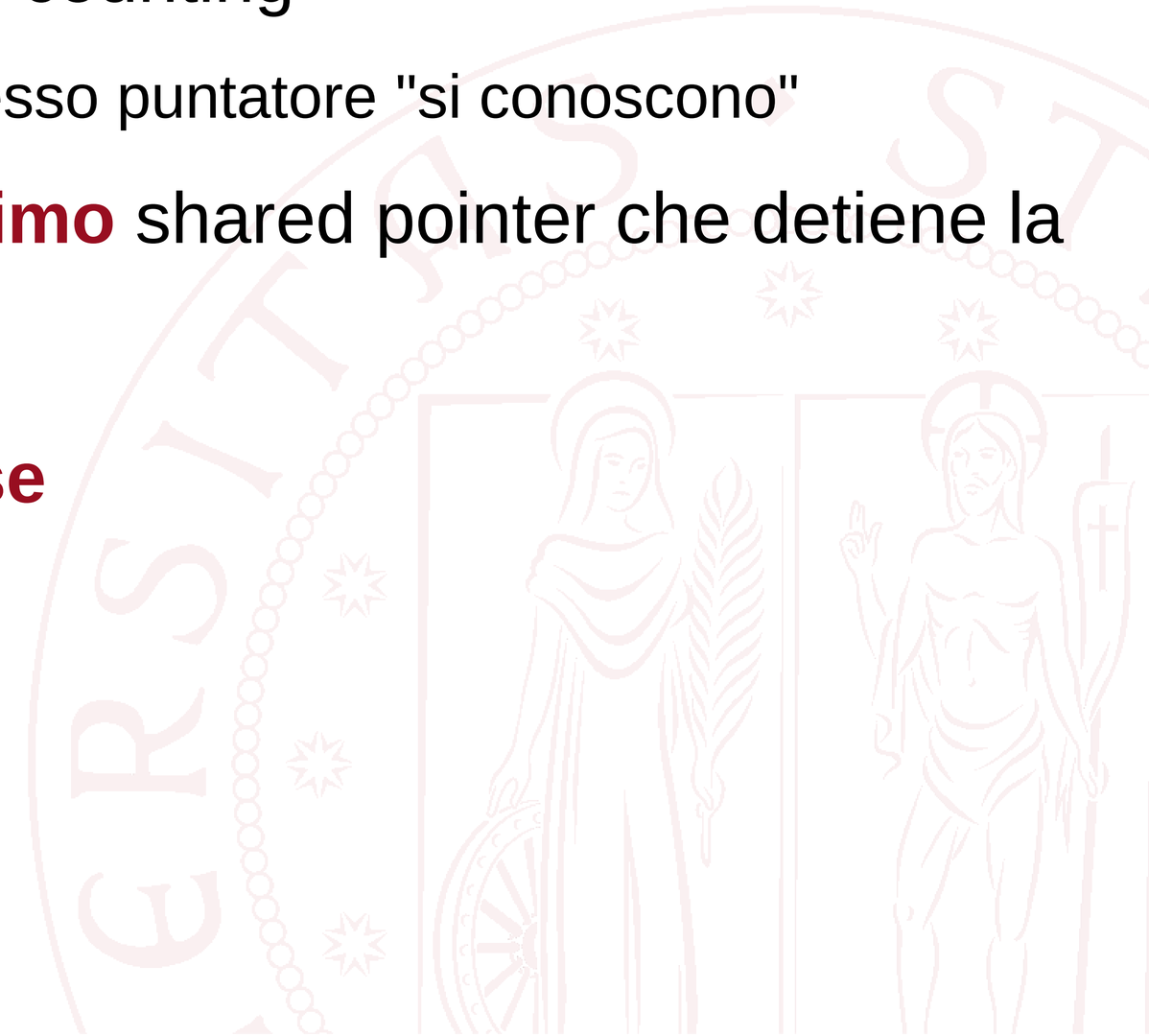
`std::shared_ptr`

- Lo shared pointer rimuove i vincoli di `std::unique_ptr`
- Uno `std::shared_ptr`:
  - Può essere copiato
  - Può essere condiviso – molti `std::shared_ptr` possono detenere lo stesso puntatore



`std::shared_ptr`

- Questo è possibile tramite il reference counting
  - `std::shared_ptr` che detengono lo stesso puntatore "si conoscono"
- La memoria è deallocata quando **l'ultimo** shared pointer che detiene la memoria è distrutto
- Questo meccanismo **consuma risorse**



Esempio

- `AllocateAndReturn`: versione con `std::shared_ptr`

```
std::shared_ptr<int> AllocateAndReturn()
{
    std::shared_ptr<int> ptr_to_return(new int(42));

    return ptr_to_return;
}
```

Esempio

- Con `std::shared_ptr` la copia è possibile

```
int main()
{
    std::shared_ptr<int> moved_ptr = AllocateAndReturn();

    // Questo è possibile!
    std::shared_ptr<int> copied_ptr = moved_ptr;

    std::cout << *moved_ptr << std::endl;

    return 0;
}
```

Recap

- Concetto di smart pointer
- `std::unique_ptr`
- `std::shared_ptr`
- Esempi – riscrivere `make_vec` con `std::unique_ptr`