13.2 – Risorse ed eccezioni

Capitoli libro:

- 19.5

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

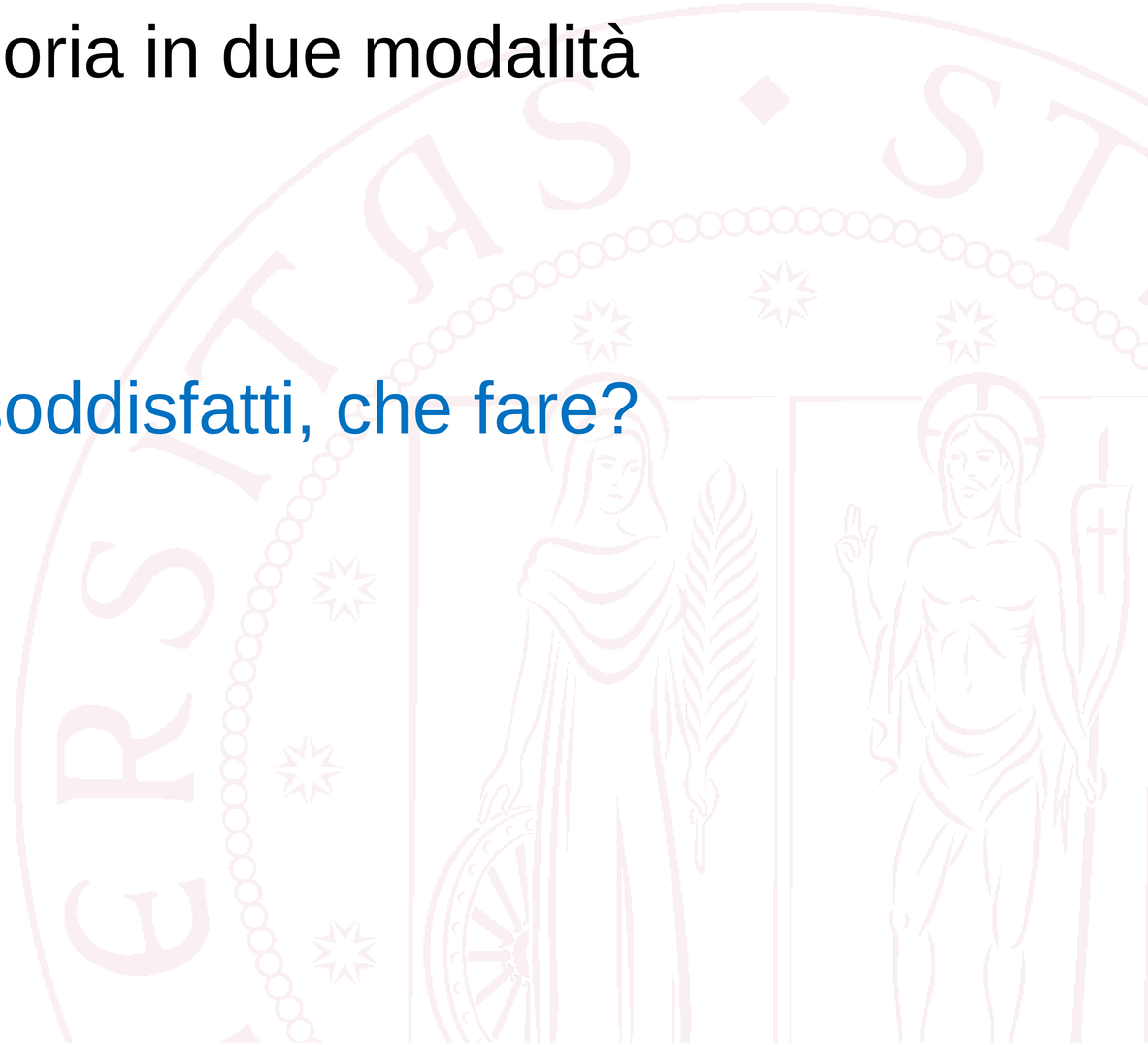
- Recap sull'acquisizione di risorse
- Gestione delle risorse in presenza di eccezioni
- Resource Acquisition Is Initialization (RAII)
- Garanzie

Risorse ed eccezioni

- In passato abbiamo analizzato la gestione delle risorse
- Ad esempio per la memoria:
  - Acquisizione
  - Utilizzo
  - Rilascio
- La gestione delle risorse può essere corrotta dalle eccezioni

Risorse ed eccezioni con `std::vector`

- Un esempio di tale situazione:
- `std::vector` può accedere alla memoria in due modalità
  - Con boundary check – funzione `at()`
  - Senza boundary check – `operator[]`
- Nel primo caso: se i vincoli non sono soddisfatti, che fare?
  - Eccezioni



`std::vector` ed eccezioni

- È normale che `std::vector` lanci eccezioni
- In questo contesto, dobbiamo assicurarci che eventuali risorse occupate siano liberate
- Le risorse possono essere:
  - Memoria
  - Lock
  - File aperti
  - Thread
  - Socket
  - Window
  - ...

Gestione delle risorse

- Prendiamo un caso di acquisizione delle risorse:

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    // ...
    delete[] p;
}
```

- Come possiamo essere sicuri che la memoria sia rilasciata? Cosa può esserci nei … ?

Gestione delle risorse

- È rilasciata in questo caso?

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    // ...
    if (x) p = q;
    // ...
    delete[] p;
}
```

- Il valore di p è cambiato

Gestione delle risorse

- È rilasciata in questo caso?

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    // ...
    if (x) return;
    // ...
    delete[] p;
}
```

- Il flusso di esecuzione non arriva alla `delete[]`

Gestione delle risorse

- È rilasciata in questo caso?

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    std::vector<int> v;
    // ...
    if (x) p[x] = v.at(x);    // at(): accesso con verifica
    // ...                    // può lanciare eccezioni
    delete[] p;
}
```

- È lanciata un'eccezione
- È un problema di eccezioni o di gestione delle risorse?

Soluzione 1

- Una possibile soluzione se è un problema di eccezioni

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    std::vector<int> v;
    // ...
    try {
        if (x) p[x] = v.at(x);
        // ...
    } catch (...) {
        delete[] p;
        throw;                  // rilancia l'eccezione
    }
    // ...
    delete[] p;
}
```

Soluzione 1

- Il codice precedente risolve il problema, però…
  - Il codice si duplica e diventa molto lungo
  - Se estendiamo questo approccio, dobbiamo gestire eccezioni:
    - A ogni accesso al vettore
    - A ogni allocazione
- Questo codice è difficile da leggere e da mantenere

Soluzione 2

- Usando solo `std::vector` (al posto dell'allocazione dinamica) il problema è risolto

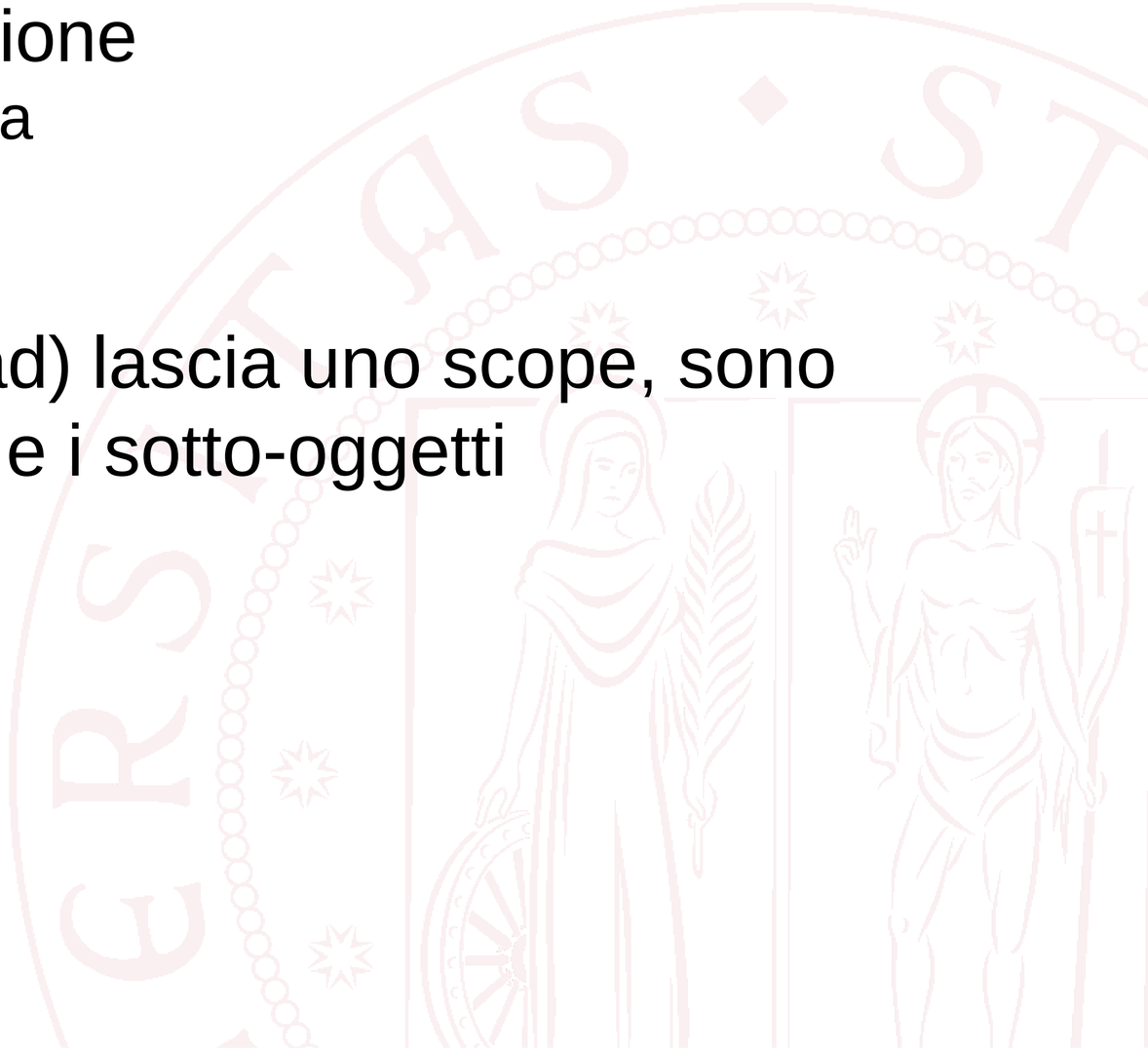
```
void f(std::vector<int>& v, int s)
{
    std::vector<int> p(s);
    std::vector<int> q(s);
    // ...
}
```

- Perché è risolto? Chi lo risolve e in che modo?

Liberare risorse

- Il problema è risolto perché la memoria è acquisita in fase di costruzione e liberata in fase di distruzione
  - Costruttori e distruttore risolvono il problema
  - Lo risolvono in maniera **semplice**

- Quando un flusso di esecuzione (thread) lascia uno scope, sono **invocati i distruttori** di tutti gli oggetti e i sotto-oggetti

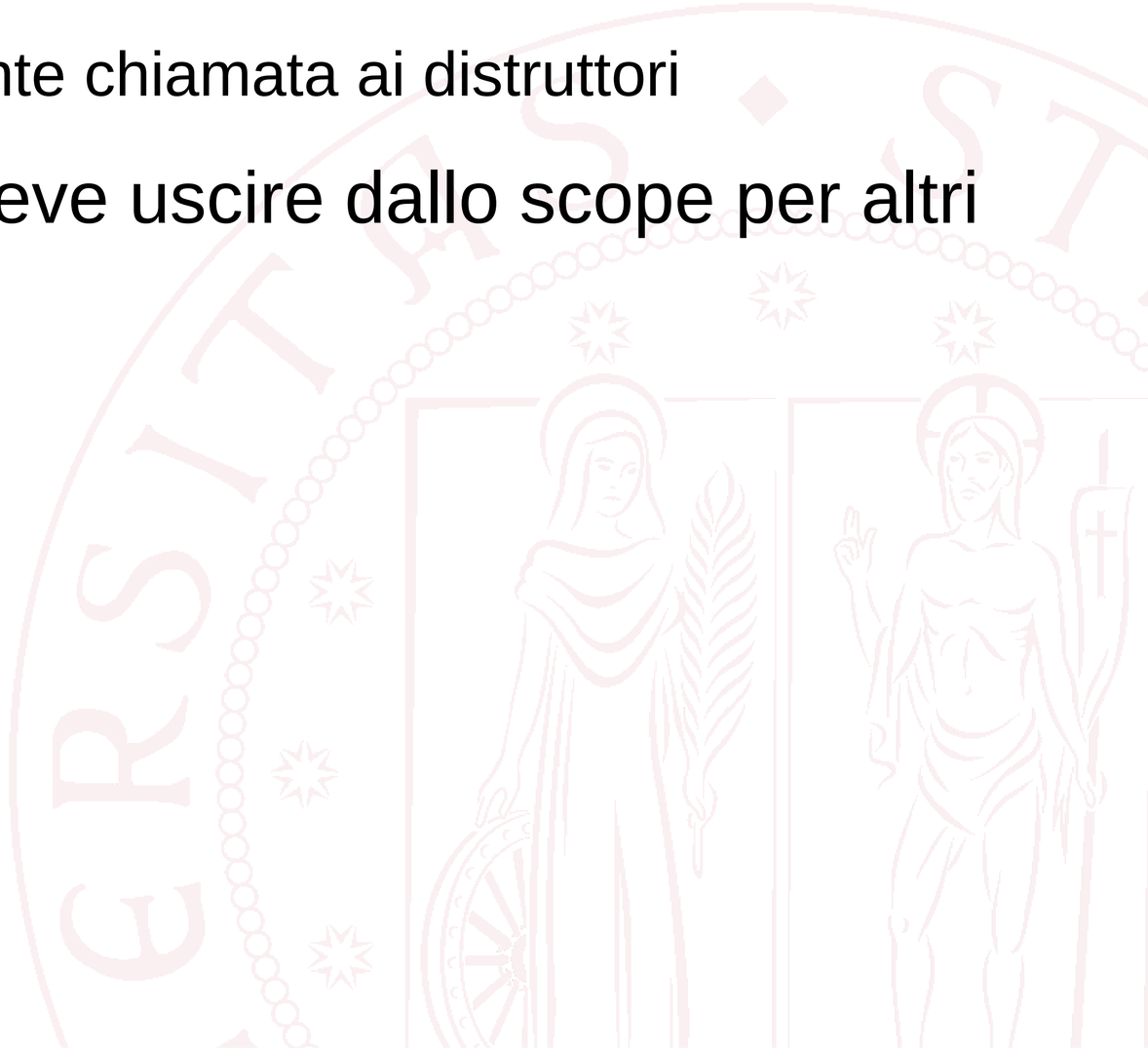


Resource Acquisition Is Initialization (RAII)

- Occupare le risorse nel costruttore e liberarle nel distruttore risolve questi problemi
- Questa tecnica prende il nome di **Resource Acquisition Is Initialization (RAII)**
- È un'idea generale: vale anche per
  - socket
  - I/O buffer (inclusi i file)
  - ...

RAII e scope

- Il meccanismo visto prima si basa sull'uscita dallo scope
  - L'uscita dallo scope libera le risorse mediante chiamata ai distruttori
- Cosa succede se il flusso del codice deve uscire dallo scope per altri motivi, ma non per liberare le risorse?



RAII e scope

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

- A volte desideriamo "far uscire" gli oggetti dallo scope
  - Caso tipico: funzione che costruisce un oggetto grande e lo ritorna usando un puntatore

RAII e scope

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

dannoso e sbagliato

Usato come esempio perché il riempimento può causare un'eccezione

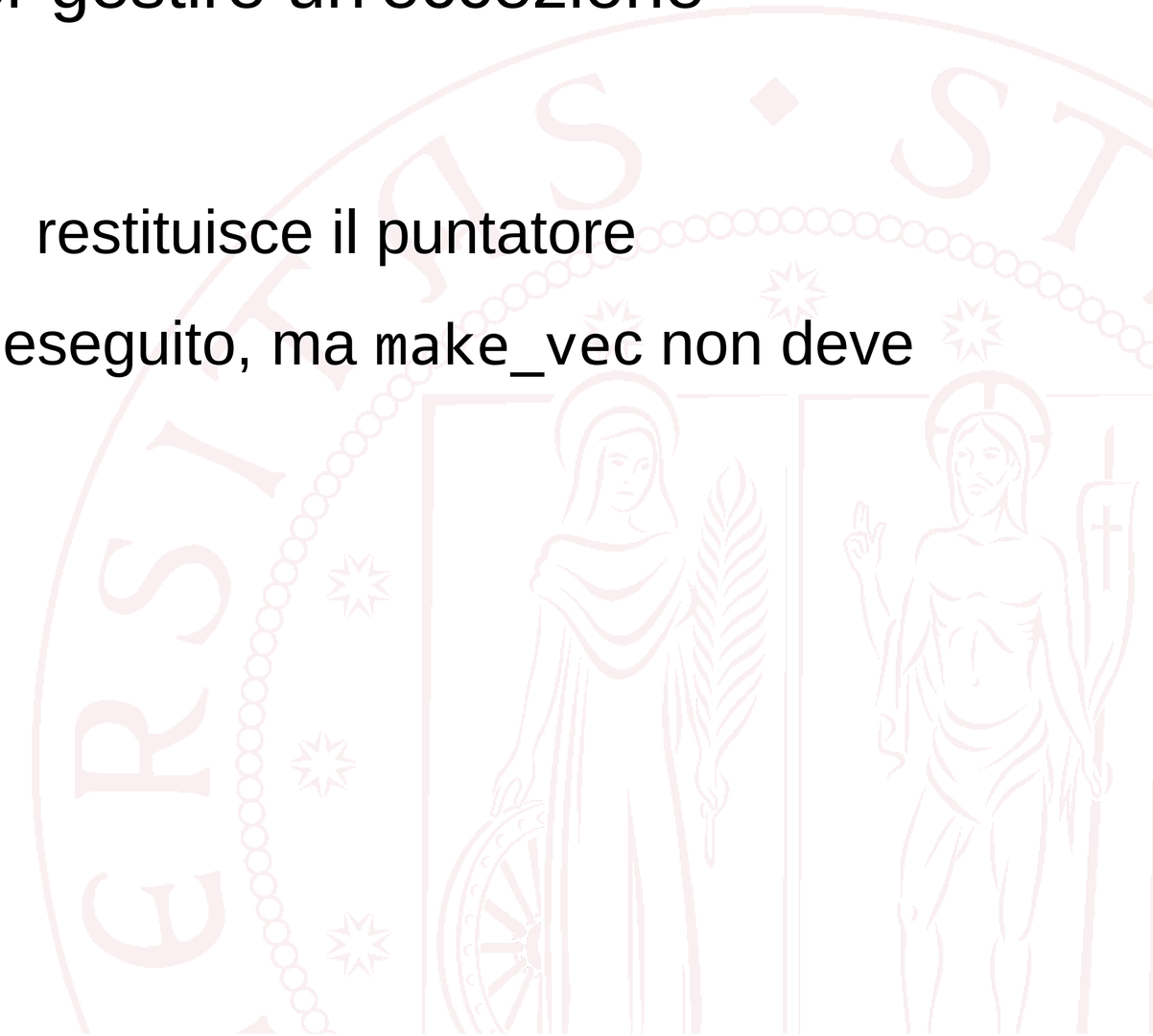
RAII e scope

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

- Cosa succede se un'eccezione è lanciata durante il riempimento?
- Caveat: `p` deve essere liberato dal chiamante

RAII e scope

- La funzione `make_vec` potrebbe dover gestire un'eccezione
- Comportamento desiderato:
  - Se non sono lanciate eccezioni, `make_vec` restituisce il puntatore
  - Se sono lanciate eccezioni, `return` non è eseguito, ma `make_vec` non deve comunque causare memory leak



RAII e scope

Implementazione di tale comportamento:

```
std::vector<int>* make_vec()
{
    std::vector<int>* p = new std::vector<int>;
    try {
    // ... riempimento del vettore con i dati – può
    // ritornare un'eccezione!
    }
    catch(...) {
        delete p;
        throw;        // rilancia l'eccezione
    }
    return p;
}
```

Garanzie

- La tecnica vista è un pattern ricorrente che prende il nome di basic guarantee
- **Basic guarantee:** il blocco `try/catch` fa sì che `make_vec()` funziona, oppure lancia un'eccezione e non crea leak
  - Necessaria per tutto il codice che deve gestire eccezioni che potrebbero essere lanciate
  - STL fornisce la basic guarantee

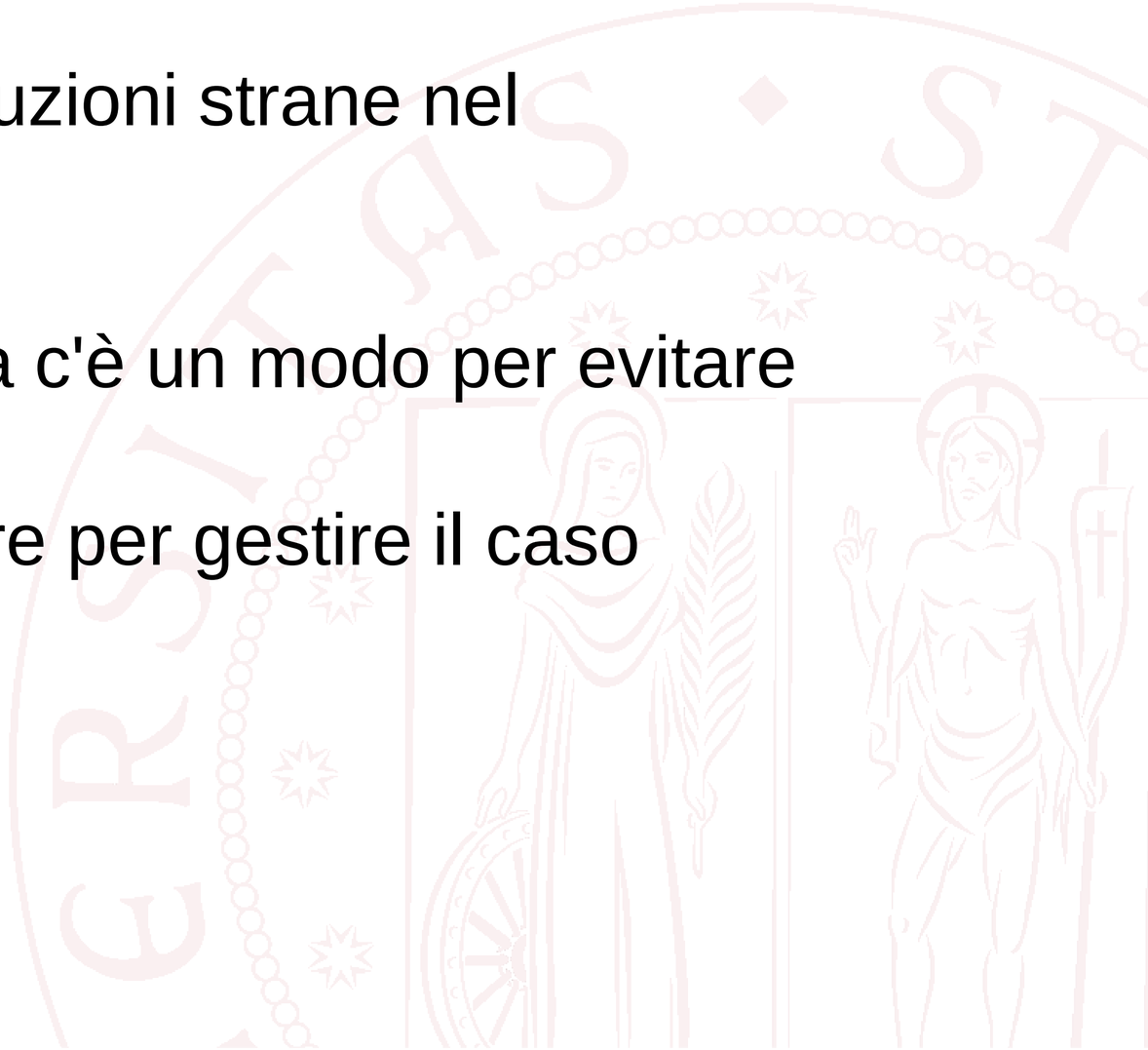
Garanzie

- **Strong guarantee:** la funzione rispetta la basic guarantee, e in più tutti i valori osservabili (valori non locali alle funzioni) sono gli stessi che erano presenti prima della chiamata, anche in caso di fallimento
- **No-throw guarantee:** la funzione non lancia eccezioni

Garanzie di `make_vec()`

- **Che garanzia offre `make_vec()`?**
  - Sicuramente la basic
  - Anche la strong, se non ci sono istruzioni strane nel riempimento del vettore

- La funzione `make_vec()` funziona, ma c'è un modo per evitare il `try/catch`?
- È "brutto": ci obbliga a scrivere software per gestire il caso particolare



Uscita dallo scope e deallocazione

```
std::vector<int>* make_vec()
{
    std::vector<int>* p = new std::vector<int>;
    try {
    // ... riempimento del vettore con i dati – può
    // ritornare un'eccezione!
    }
    catch(...) {
        delete p;
        throw;        // rilancia l'eccezione
    }
    return p;
}
```

- Risolviamo il problema precedente se riusciamo a collegare:
  - Uscita dallo scope
  - Deallocazione
- Esistono strumenti che gestiscono questa situazione: **smart pointer**

Recap

- Gestire le risorse in presenza di eccezioni
- RAII
- Garanzie
  - Basic guarantee
  - Strong guarantee
  - No-throw guarantee