6.1 – Allocare la memoria usando il free store

Libro di testo:

Capitoli 17.4, 17.4.1-5

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Come creare una classe `std::vector`?
- Acquisizione di memoria dal free store
- Puntatori e free store
- Null pointer
- Garbage

Creare una classe `std::vector`

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

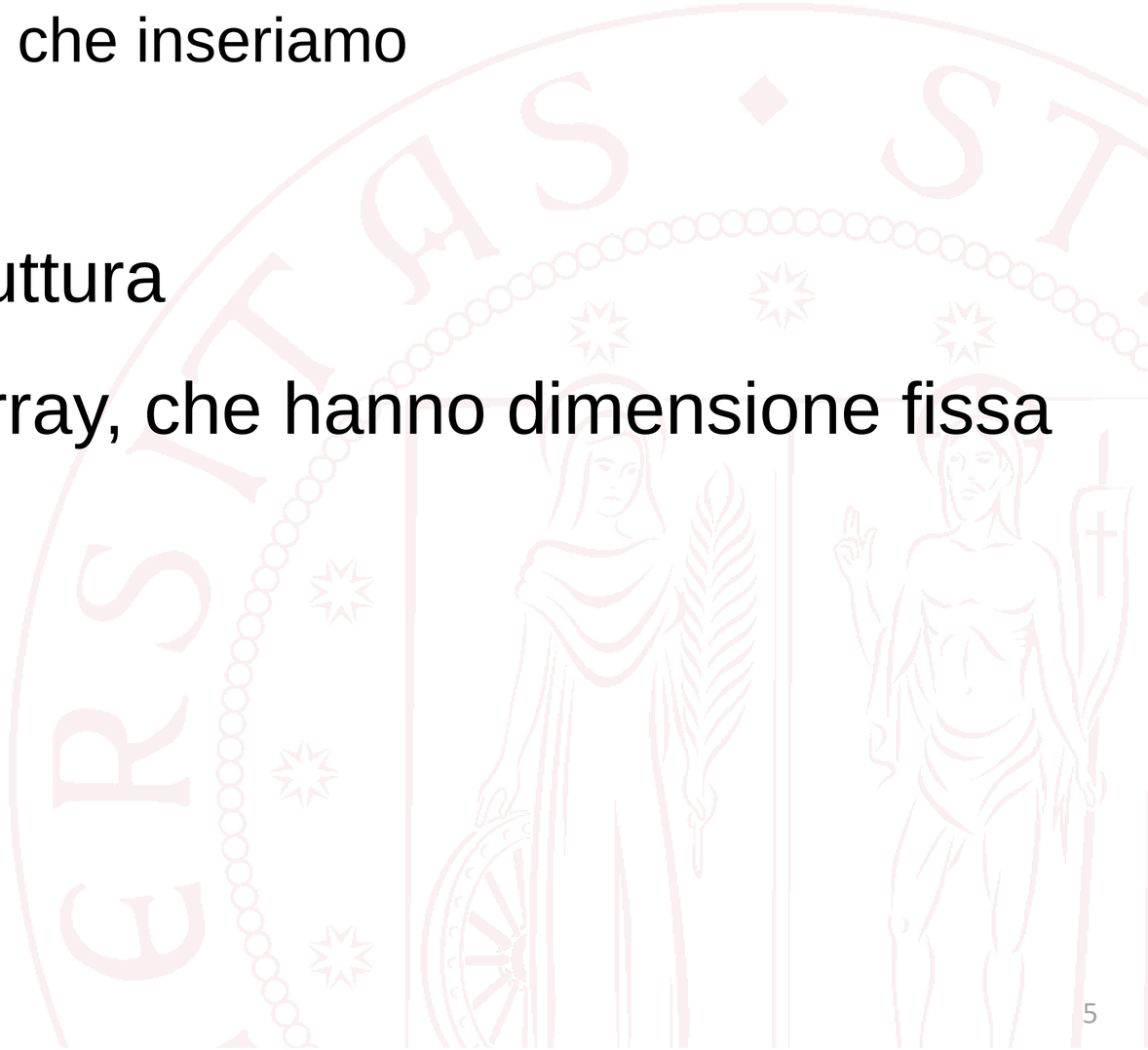
Richiamo: `std::vector`

- Abbiamo già usato gli `std::vector`
  - Sequenza di oggetti uguali
  - Riferimento agli elementi: `[]`
  - Aggiunta di un elemento: `push_back()`
  - Numero di elementi: `size()`
  - Accesso con verifica (se richiesto)

- Altri container utili:
  - `std::string`
  - `std::list`
  - `std::map`
  - …

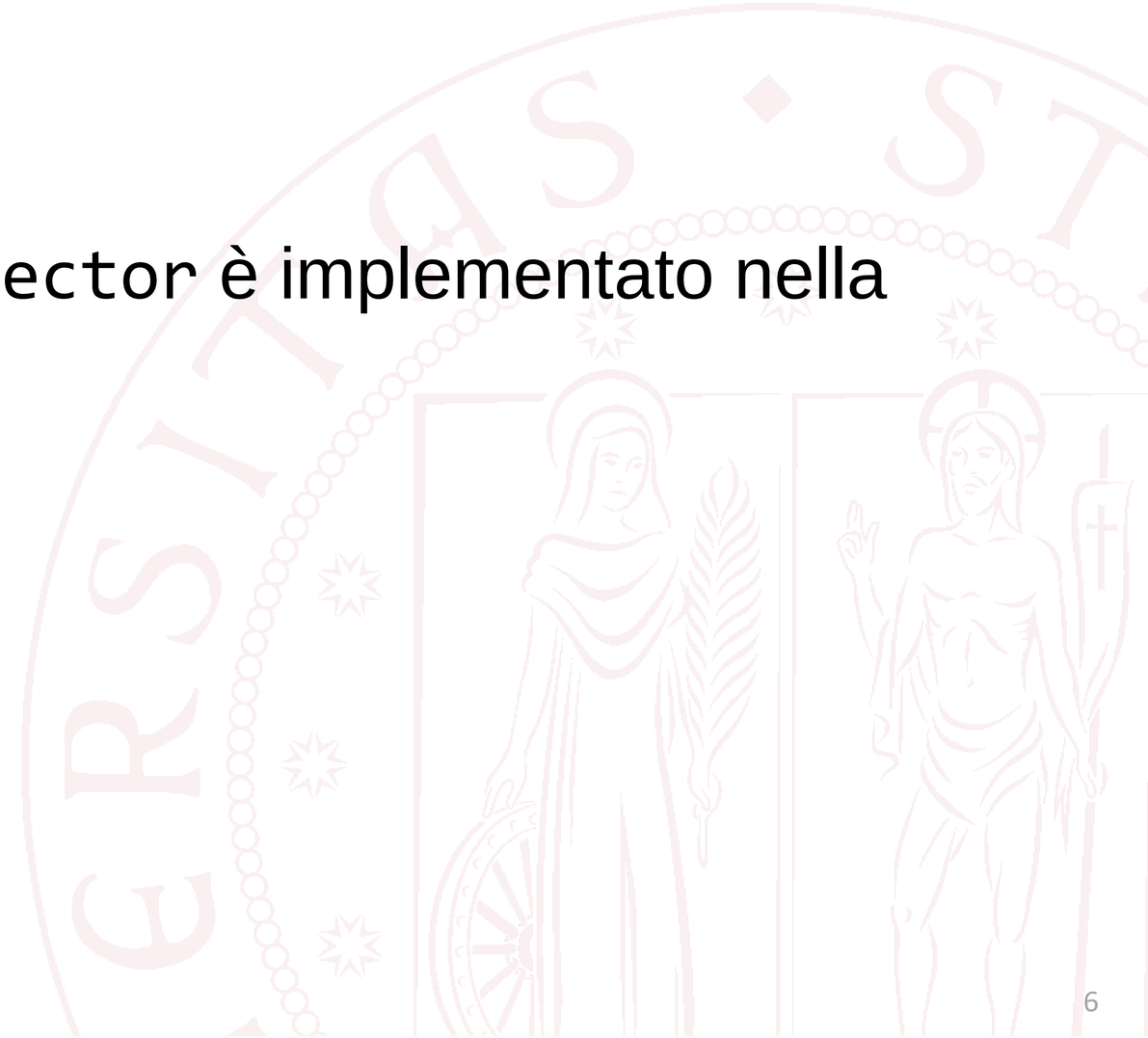
`std::vector`

- `std::vector` ha un comportamento intelligente
  - Cambia dimensione per ospitare gli elementi che inseriamo
  - Conosce la propria dimensione
- Non è supportato in HW – una sovrastruttura
- Non può essere implementato con gli array, che hanno dimensione fissa



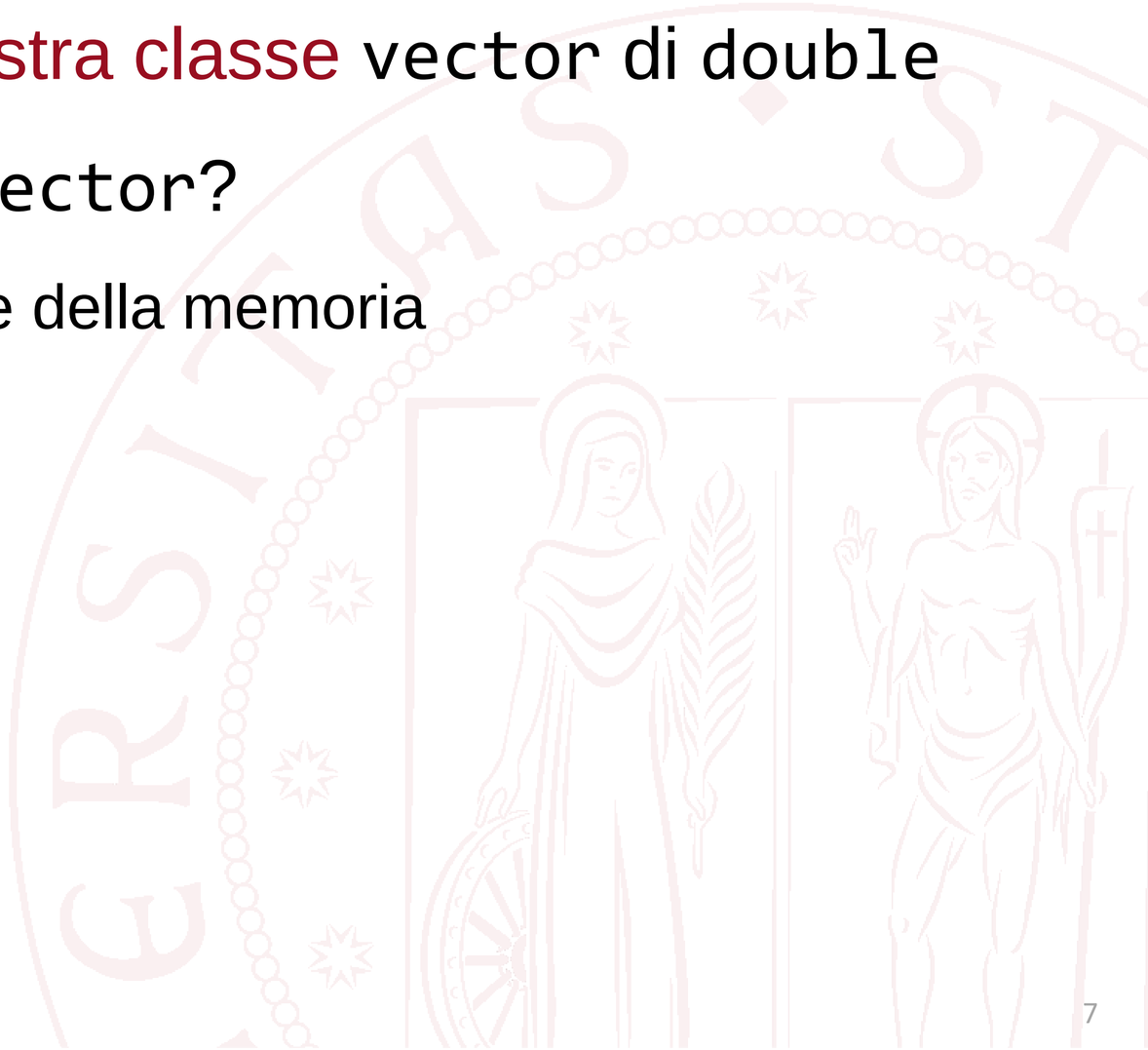
`std::vector`

- A basso livello (HW) gli strumenti a disposizione sono molto meno evoluti:
  - Memoria
  - Indirizzi di memoria
- Il comportamento intelligente di `std::vector` è implementato nella libreria standard!



Implementare una classe `vector`

- Come possiamo implementare una nostra classe `vector` di `double` con le stesse caratteristiche di `std::vector`?
  - Un compito con cui impariamo la gestione della memoria



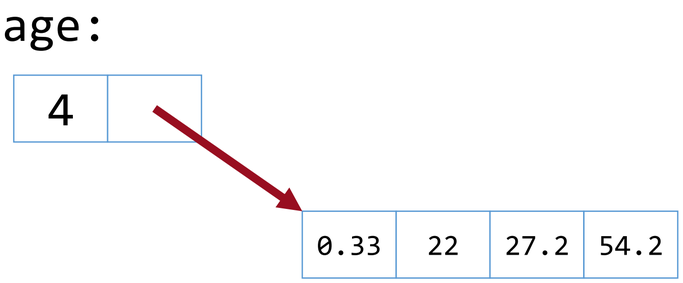
Struttura di un `vector`

- Consideriamo un uso semplice:

```
vector age(4);

age[0] = 0.33;
age[1] = 22;
age[2] = 27.2;
age[3] = 54.2;
```

Gli elementi di un vettore hanno indici da `0` a `size-1`

- Il numero di elementi di un vettore è chiamato `size`
- Quindi il nostro vettore `age` ha `size 4`
  - Da realizzare con questa struttura:



- Perché così?
  - Come lo si programma?
  - Come rendere variabile il numero di elementi?

Struttura dinamica del `vector`

- **Come lo si programma?**
- Il `vector` è una struttura dinamica
  - Non posso usare una struttura statica come la classe seguente:

```
class vector {
    int size;
    double age0, age1, age2, age3;
};
```

Struttura dinamica del `vector`

- Per poter realizzare una struttura dinamica ho bisogno di:
  - Un modo per gestire insiemi di elementi
  - Crearli, modificarli, eliminarli
  - Un modo per riferirmi a tali insiemi
- Per riferirmi a tali insiemi usiamo i **puntatori**

`vector` con i puntatori

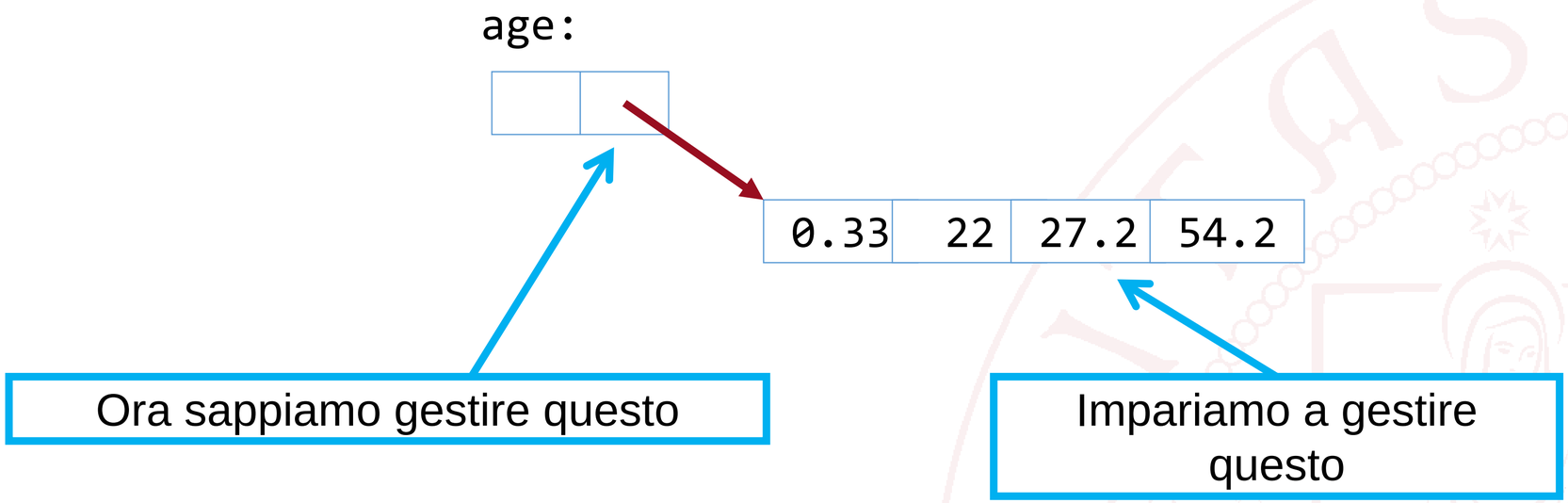
- Il nostro `vector` può essere creato così:

```
class vector {
    int sz;
    double *elem;

    public:
        vector(int s);                                  // crea un vettore di
                                                        // s double e
                                                        // fa sì che elem
                                                        // punti ad esso

        int size() const { return sz; }
};
```

- Il puntatore deve essere usato in modo da gestire un buffer

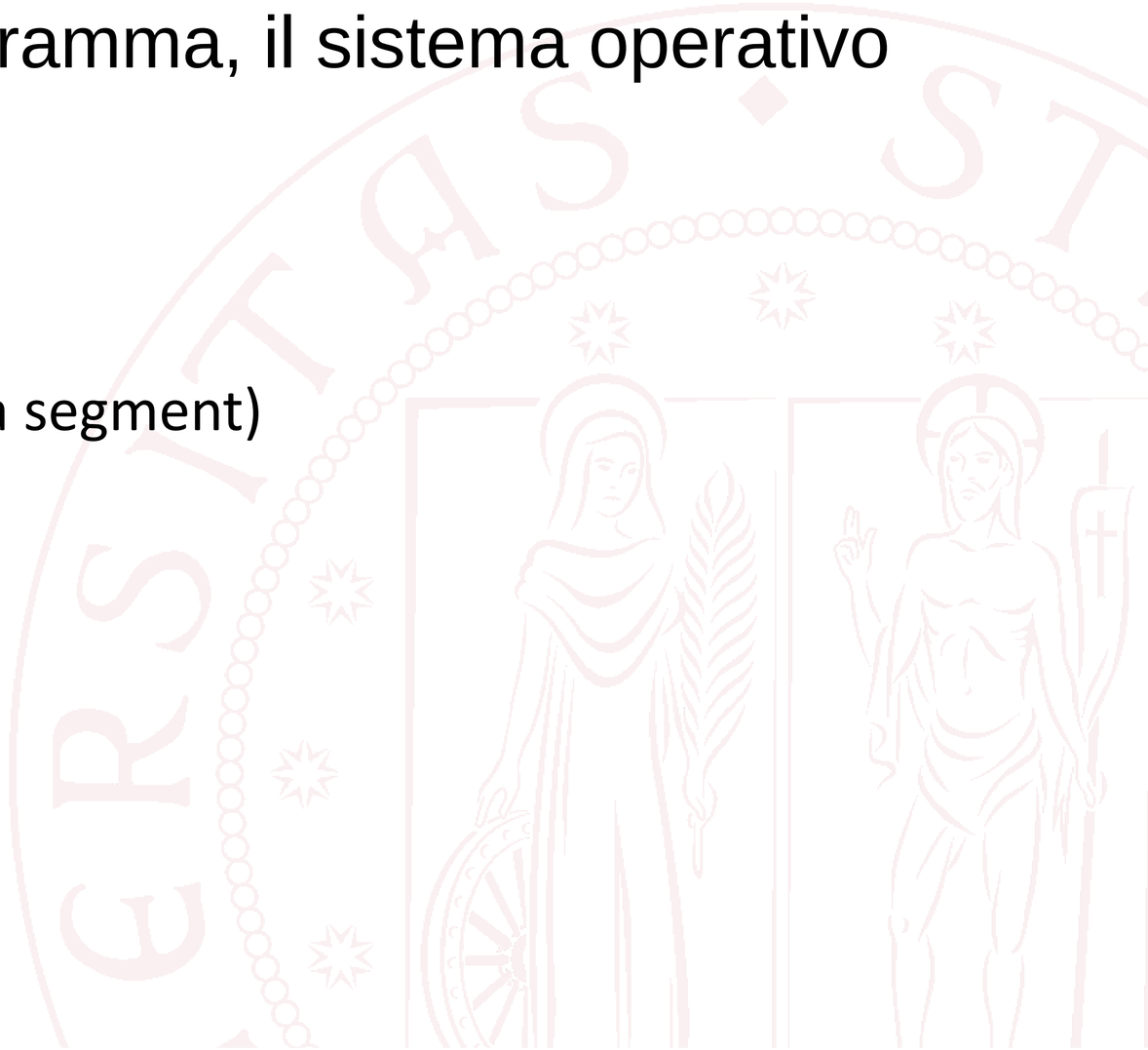
Gestione della memoria dei `vector`

- Riprendiamo l'implementazione di `vector`



Free store

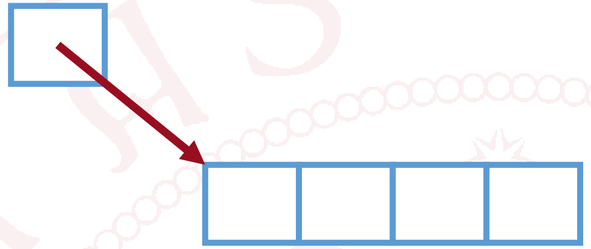
- Come rendere variabile il numero di elementi?
- Quando manda in esecuzione un programma, il sistema operativo riserva spazio per:
  - Codice (text)
  - Variabili globali (initialized/uninitialized data segment)
  - Chiamate a funzione (stack)
- Il resto è disponibile: **Free store**
  - **Heap** è (quasi) un sinonimo



Allocazione dinamica

- Come accedere al free store?
  - Usando **new** e i **puntatori**

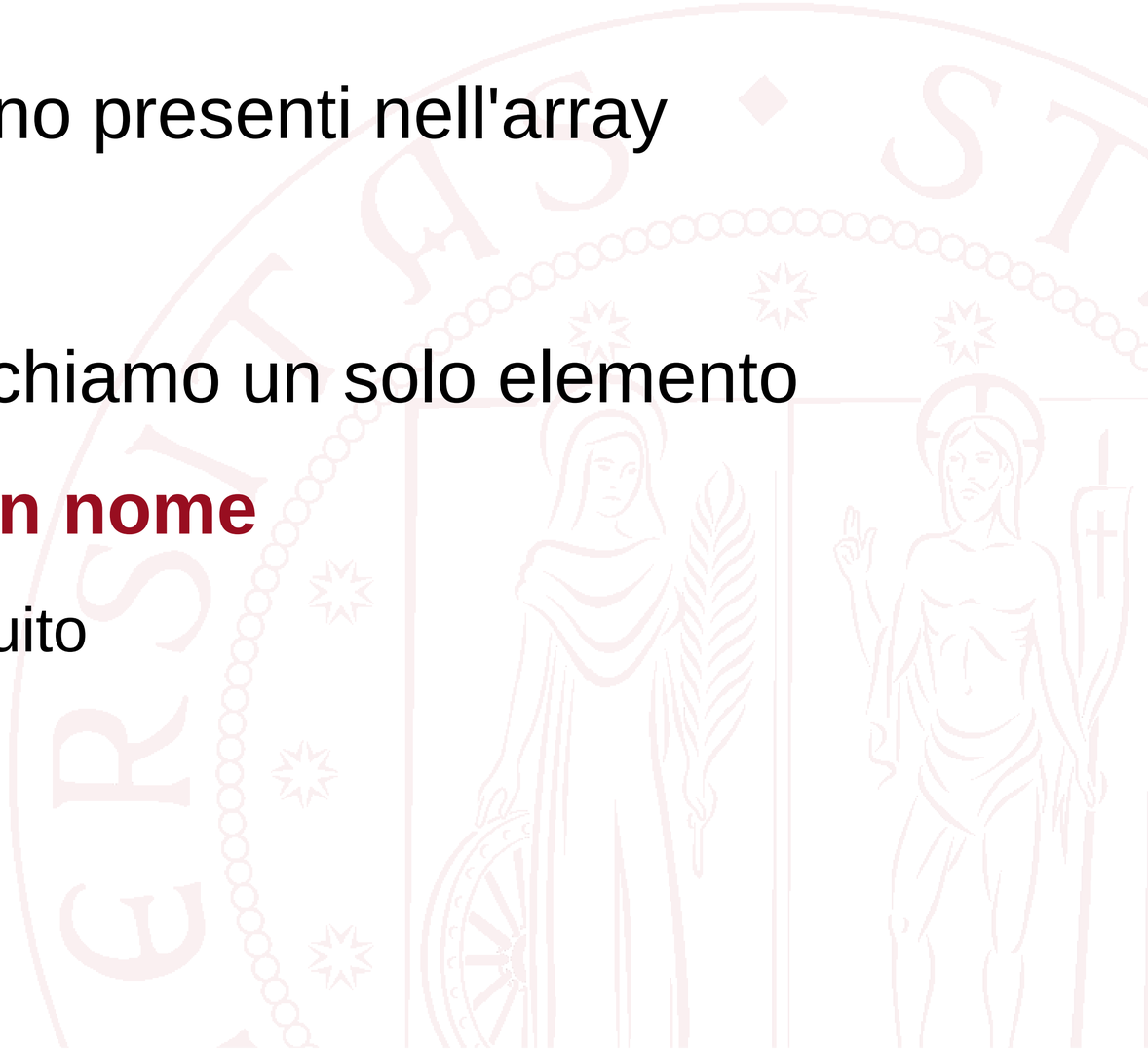
```
double* p = new double[4];
```



- new usa il free store
  - Ritorna il puntatore al primo elemento se è un array
- Questo procedimento prende il nome di **allocazione dinamica della memoria**

Puntatori e Free Store

- `new` ritorna elementi singoli o array
- Un puntatore **non sa** quanti oggetti sono presenti nell'array ritornato da `new`
- `new` ritorna un puntatore anche se allochiamo un solo elemento
- La memoria ritornata da `new` **non ha un nome**
  - Memoria gestita tramite il puntatore restituito

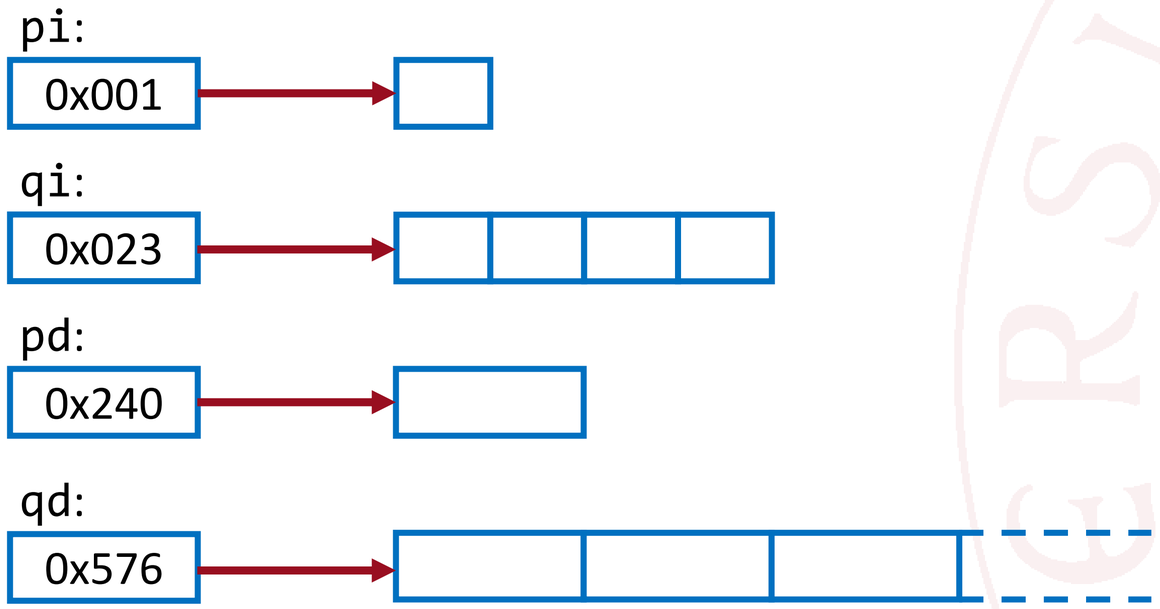


Puntatori e Free Store

Esempi:

```
int* pi = new int;
int* qi = new int[4];

double* pd = new double;
double* qd = new double[n];
```

Può essere una variabile



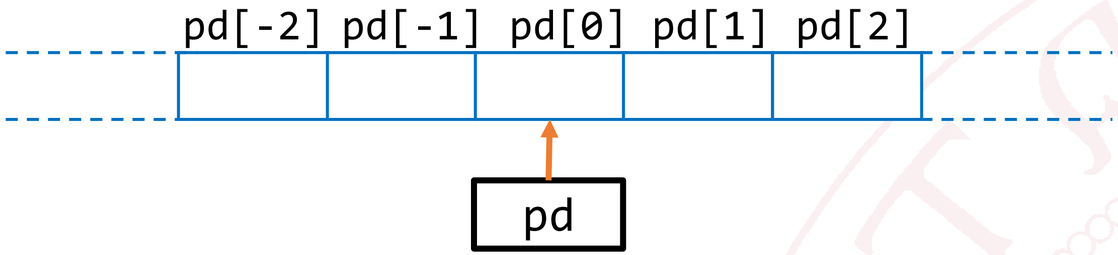
Accesso agli elementi

```
double* p = new double[4];
double x = *p;                  // primo oggetto puntato
double y = p[2];                // terzo oggetto
```

- Nella gestione della memoria dinamica usiamo spesso:
  - Il nesso tra puntatori e array
  - L'aritmetica dei puntatori
- Ricordiamo che:
  - `*p` e `p[0]` sono equivalenti
  - Applicare `[]` a un puntatore equivale a vedere la memoria come un array

Range e range check

A questo livello non esiste controllo sui range!




```
pd[2] = 2.2;        // ok!
pd[4] = 4.4;        // ok!
pd[-3] = -3.3;      // ok, ma...
```

"Il mio programma termina misteriosamente…"

- Grave problema
- Esecuzioni diverse possono dare sintomi diversi
- Bug molto difficili da gestire!

Inizializzazione dei puntatori

```
double* p0;                          // grosso problema!

double* p1 = new double;             // double non inizializzato

double* p2 = new double {5.5};       // double inizializzato

double* p3 = new double[5];          // array non inizializzato
```

- La memoria allocata con `new` non è inizializzata per i tipi built-in
- Per ovviare:

```
double* p4 = new double[5] {0, 1, 2, 3, 4}; // array init

double* p5 = new double[] {0, 1, 2, 3, 4};  // dimensione
                                            // dedotta
```

Inizializzazione con UDT

```
X* px1 = new X;

X* px2 = new X[17];
```

Chiamata al costruttore di default

- Se il costruttore di default non esiste:
  (ad esempio, esiste solo un costruttore che accetta un `int`)

```
Y* py1 = new Y;              // errore

Y* py2 = new Y[17];          // errore

Y* py3 = new Y{13};          // ok

Y* py4 = new Y[17] {1, 2, 3, 4, ..., 16};   // ok
```

Null pointer

- È molto comodo avere un valore non valido per segnalare un puntatore non inizializzato

```
double* p0 = nullptr;
```

- `nullptr` è C++11
  - Altrimenti: `NULL` oppure `0`

Controllo di validità

```
if (p0 != nullptr) {/* ... */}

if (p0) {/* ... */}
```

- Attenzione quando:
  - Il puntatore non è mai stato inizializzato
  - **Non è inizializzato a `nullptr` automaticamente!**
  - Il puntatore si riferisce a memoria non valida (liberata)

Spostamento di un puntatore

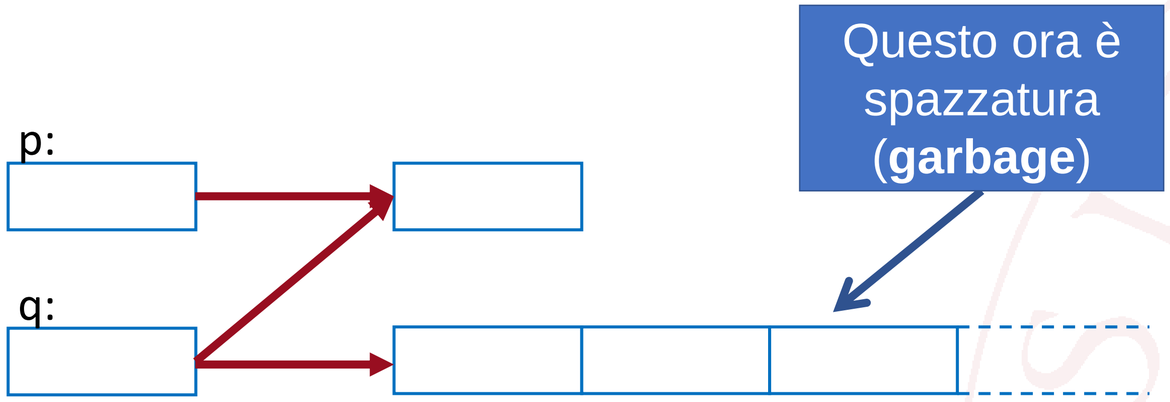
- Un puntatore può essere spostato da un oggetto a un altro
  - Differenza con le reference!

```
double* p = new double;
double* q = new double[1000];

q[700] = 7.7;          // ok
q = p;                 // attenzione!!
double d = q[700];     // attenzione!!
```

- Questo può generare grossi problemi

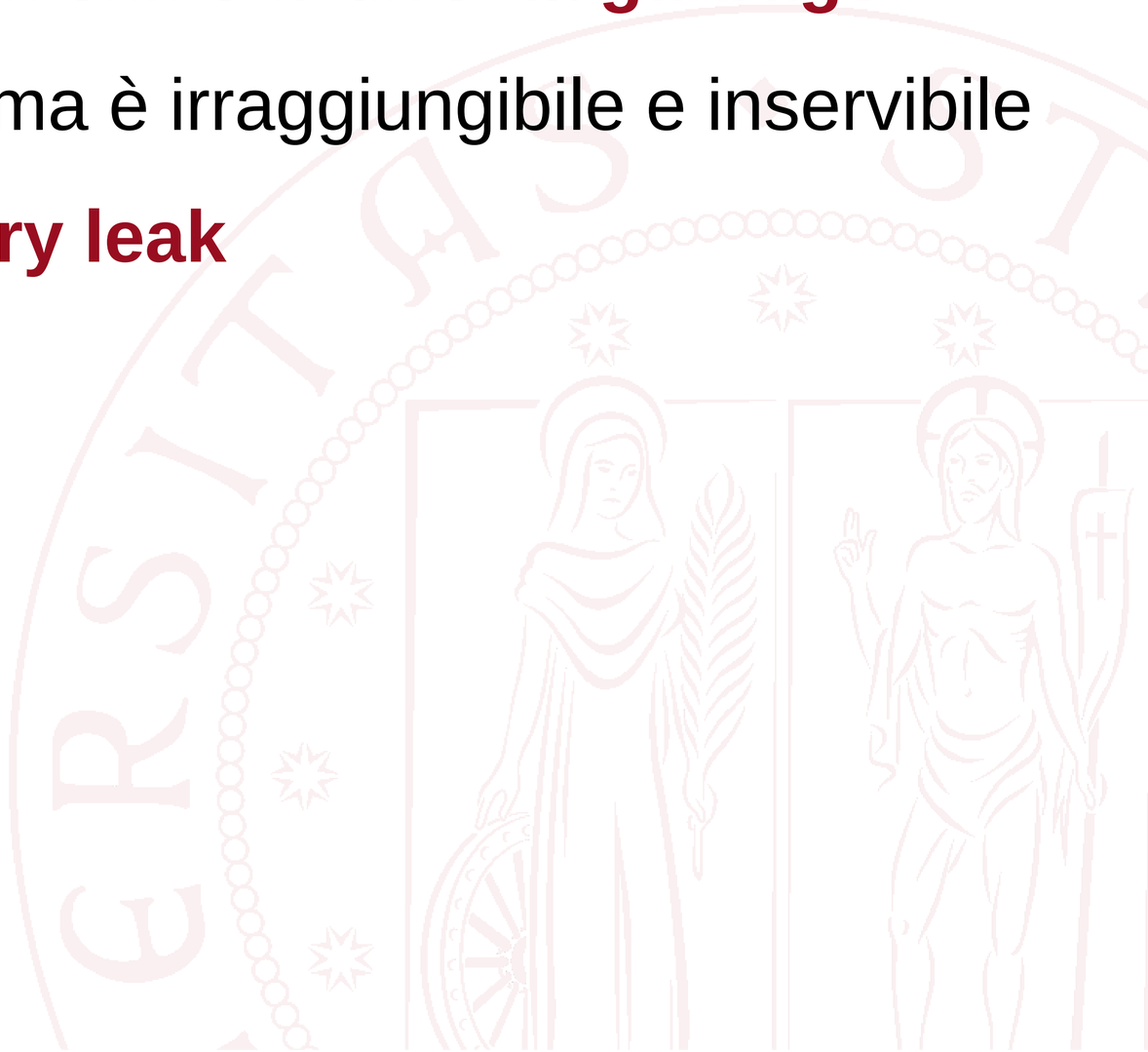
Spostamento di un puntatore

```
double* p = new double;
double* q = new double[1000];

q = p;          // attenzione!!
```



Garbage

- Un'area di memoria di cui si perdono i riferimenti diventa **garbage**
- Tale zona di memoria rimane allocata ma è irraggiungibile e inservibile
- Questo fenomeno è noto come **memory leak**
  - Può portare all'esaurimento della memoria



Recap

- Creazione di strutture dinamiche come `std::vector`
- Allocazione dinamica della memoria
  - Accesso al free store
- Accesso agli elementi e range check
- Inizializzazione
- `nullptr`
- Garbage