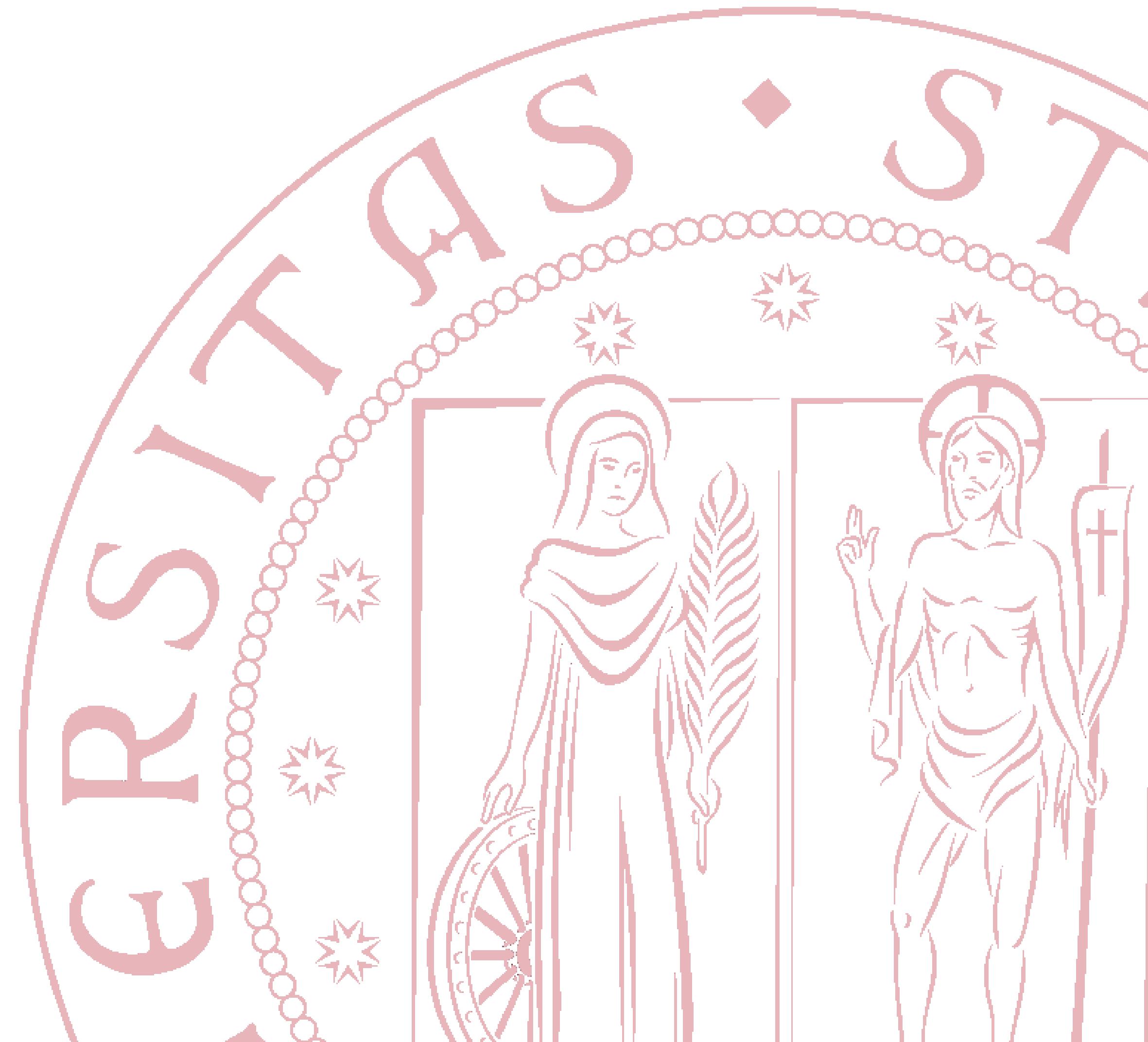


6.2 – Liberare la memoria

Libro di testo:

Capitoli 17.4.6, 17.5, 17.5.1



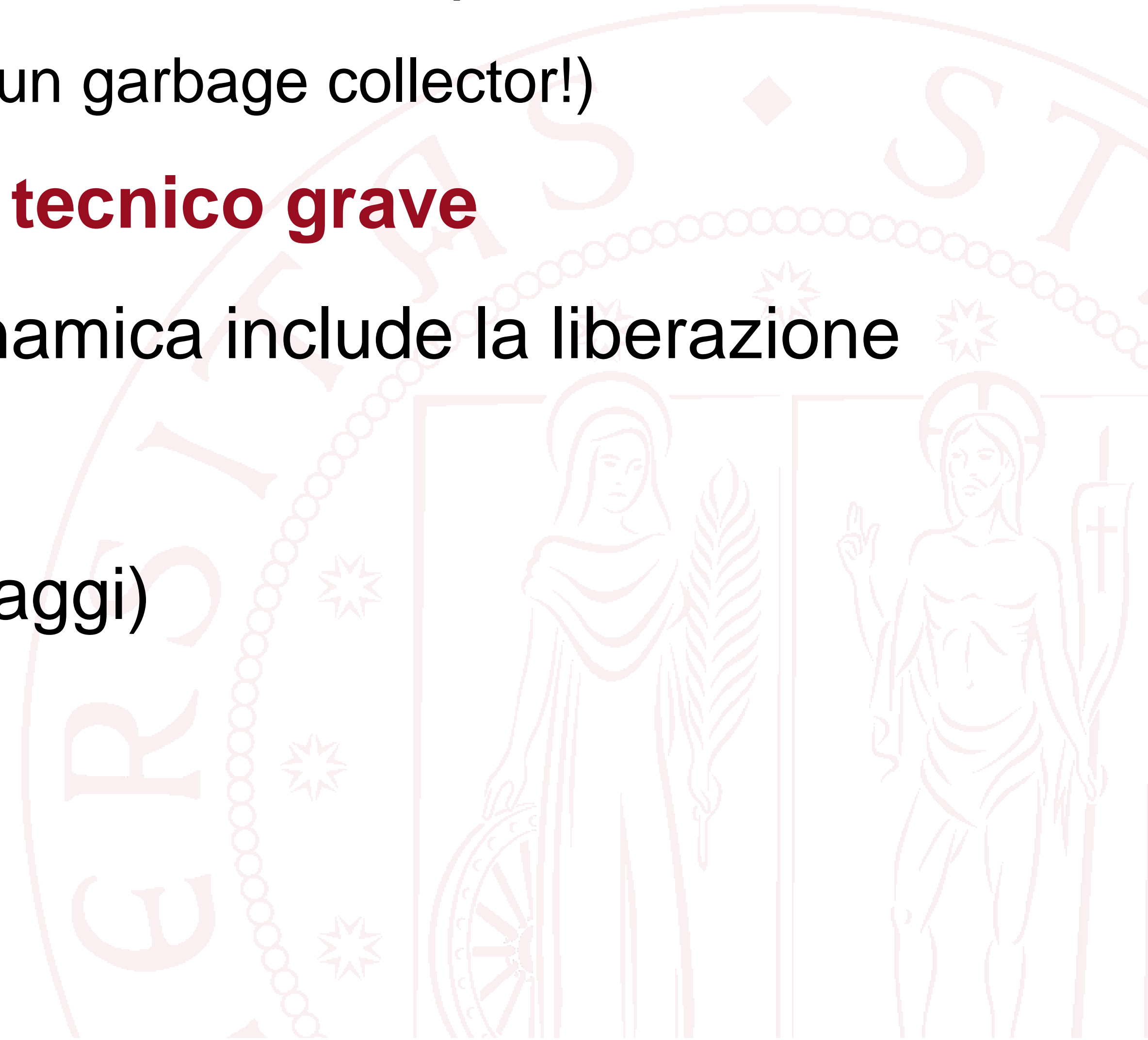
Agenda

- Memory leak e loro risoluzione
- Liberare la memoria
- Distruttore



Liberare la memoria

- È importante liberare la memoria quando non serve più
 - In C++ nessuno lo fa al posto nostro (nessun garbage collector!)
- La produzione di garbage è un **errore tecnico grave**
- La gestione corretta della memoria dinamica include la liberazione della memoria
- Forte differenza con Java (e altri linguaggi)



Liberare la memoria

- Perché dobbiamo liberare la memoria manualmente?



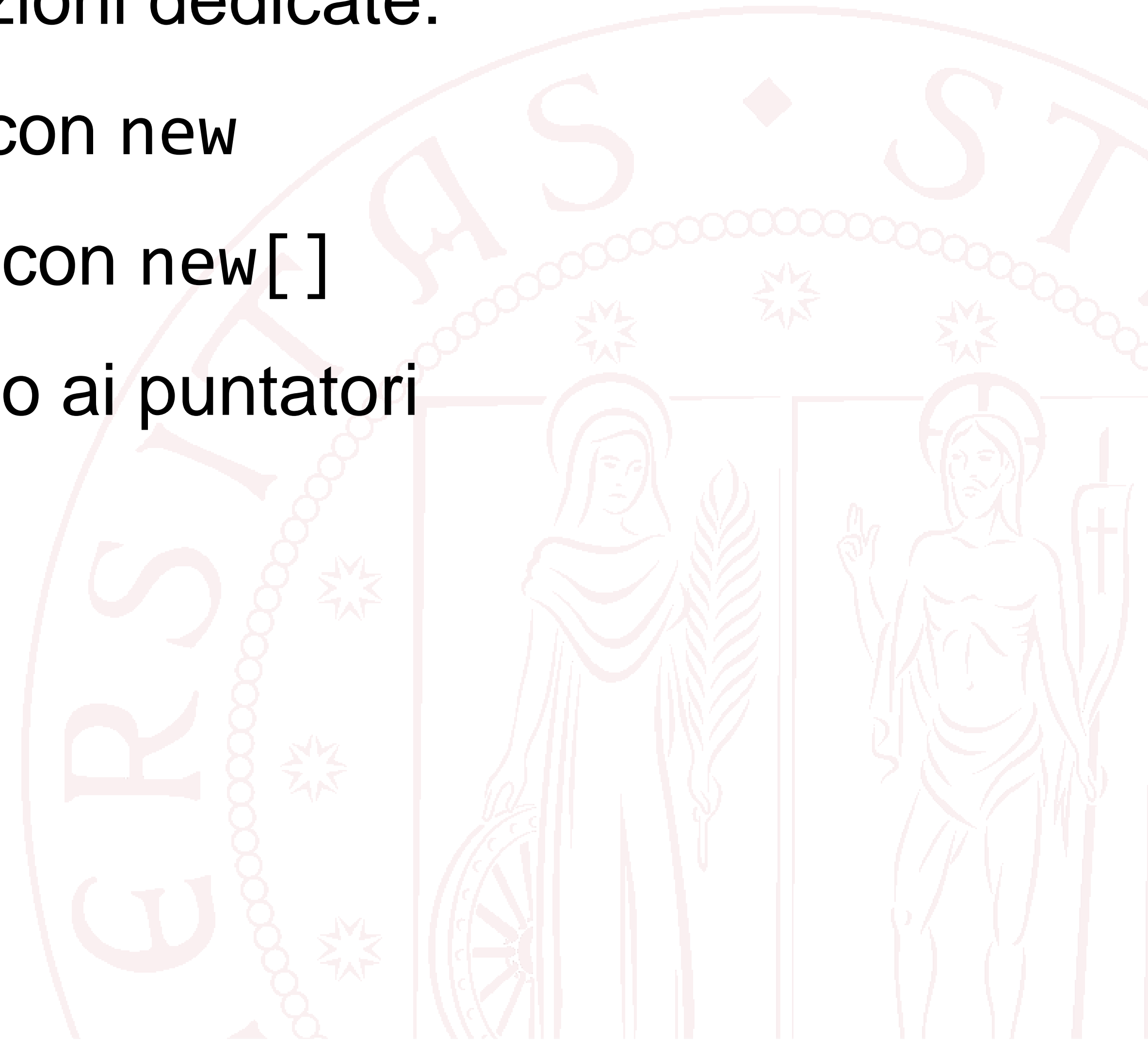
Liberare la memoria

- Perché dobbiamo liberare la memoria manualmente?
- Due motivi importanti:
 - Efficienza: il garbage collector **impiega molte risorse**
 - Deve capire quale memoria è ancora raggiungibile
 - Controllo: decidiamo quando la memoria è rilasciata
 - perciò di nuovo disponibile per altro



Liberare la memoria

- Per liberare la memoria si usano istruzioni dedicate:
 - `delete` - se memoria allocata con `new`
 - `delete[]` - se memoria allocata con `new[]`
- Sia `delete` che `delete[]` si applicano ai puntatori



Caccia al memory leak

```
double* calc(int res_size, int max) {  
    double* p = new double[max];  
    double* res = new double[res_size];  
    return res;  
}
```

Necessario
liberare p

```
// ...
```

```
double* r = calc(100, 1000);
```

Necessario
liberare r

```
// ... ← ma nessuna deallocazione di r
```

- Dove sono i memory leak?

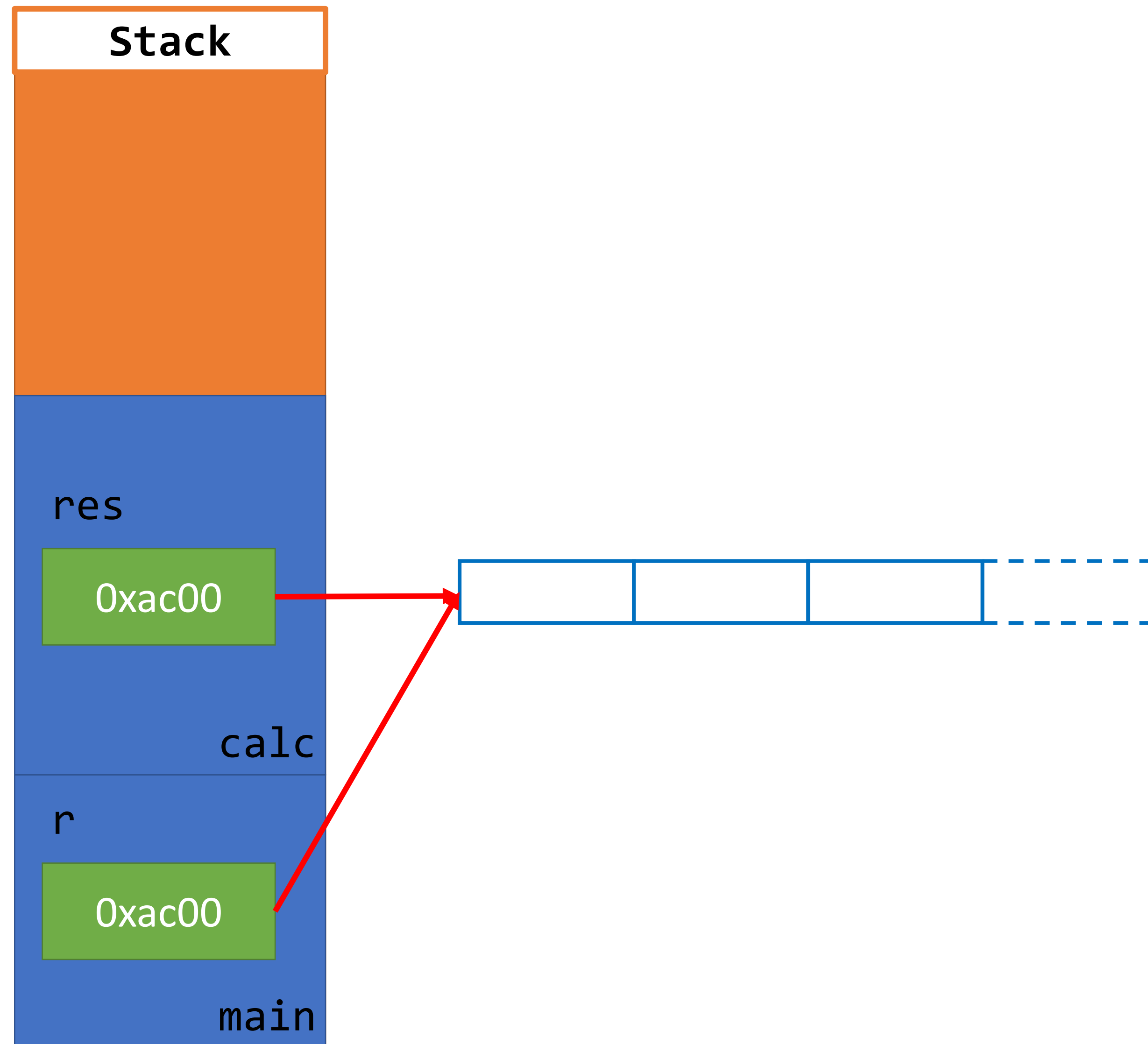
Risolvere i memory leak

```
double* calc(int res_size, int max) {  
    double* p = new double[max];  
    double* res = new double[res_size];  
    delete[] p;  
  
    return res;  
}  
  
// ...  
  
double* r = calc(100, 1000);  
  
// ...  
delete[] r;
```


Passaggio di memoria tra funzioni

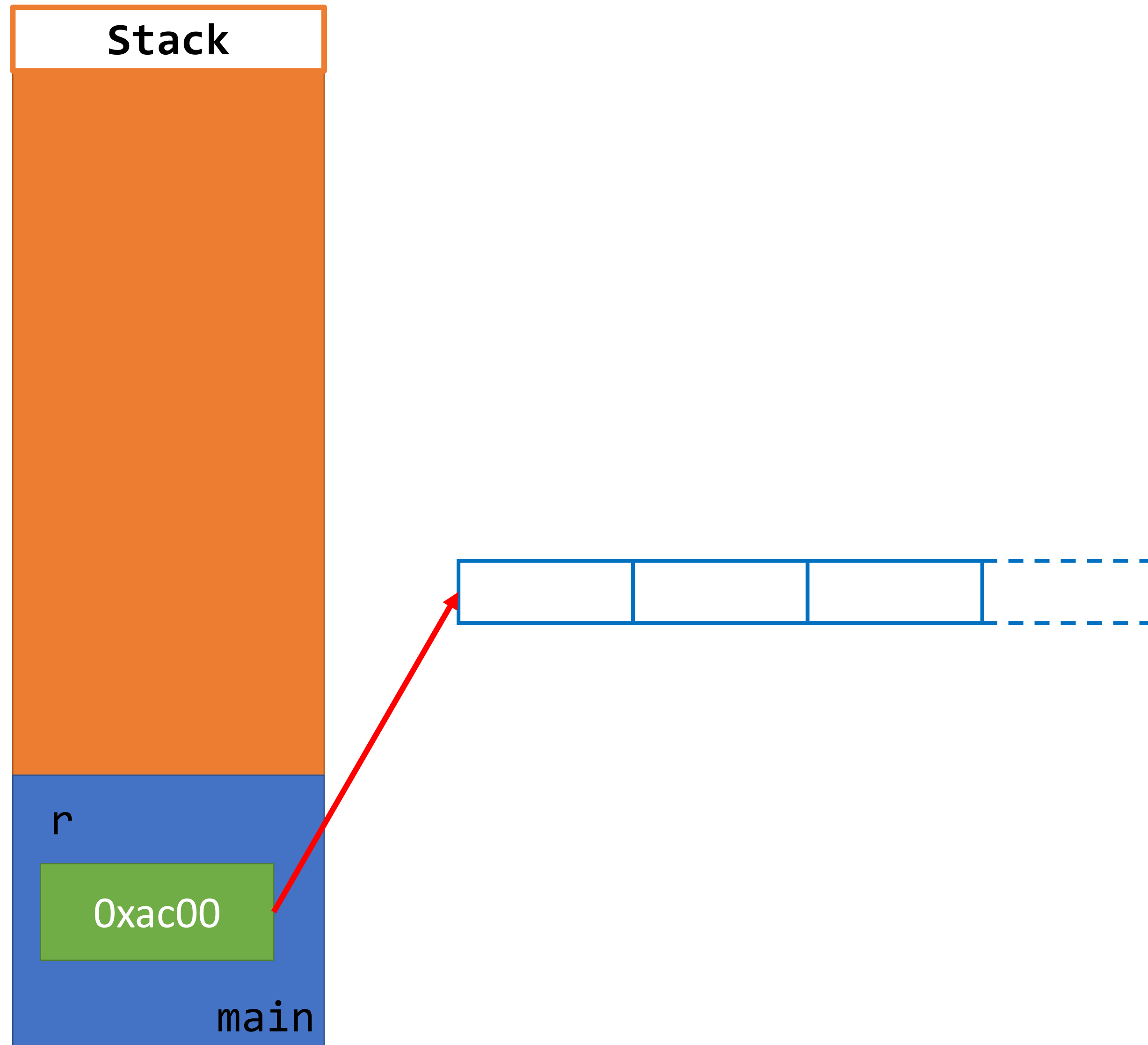
- Con l'allocazione dinamica della memoria possiamo creare un oggetto in uno scope (es., funzione) e passarlo a un altro scope (es., il chiamante)
- Nel caso precedente: `calc()` alloca `res` e lo ritorna al chiamante
 - È copiato solo il puntatore, la memoria allocata rimane tale quale

Restituire un puntatore



```
double* calc(int res_size) {  
    double* res = new double[res_size];  
    return res;  
}  
  
int main() {  
    double* r = calc(1000);  
}
```

Restituire un puntatore



```
double* calc(int res_size) {  
    double* res = new double[res_size];  
    return res;  
}  
  
int main() {  
    double* r = calc(1000);  
  
    // ...  
    delete[] r;  
}
```

Deallocazione nel chiamante

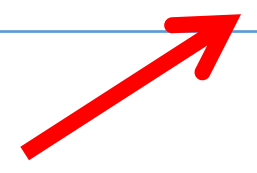
- Un altro esempio:

```
vector* f(int s) {  
    vector* p = new vector(s);  
    // fill p  
    return p;  
}  
  
void pf() {  
    vector* q = f(4);  
    // use q  
    delete q;  
}
```

Doppia cancellazione

- È un errore grave deallocare due volte la memoria

```
int* p = new int{5};  
  
delete p;  
  
// ... - ma nessun uso di p  
  
delete p;
```



- p non è più a nostra disposizione
 - Il proprietario è il free store manager
 - Potrebbe esserci un altro oggetto

Dangling pointer

- Dopo il `delete`, il puntatore mantiene lo stesso valore
- Tale valore non è più valido ed è un errore utilizzarlo
- Questa situazione prende il nome di **dangling pointer**
(*puntatore pendente/penzolante*)



Dangling pointer

- Per evitare il dangling pointer è opportuno settare il puntatore a nullptr

```
int* p = new int{5};  
//...  
delete p;  
p = nullptr;
```

Liberare la memoria con UDT

- Torniamo al nostro esempio di `vector<double>`

age:



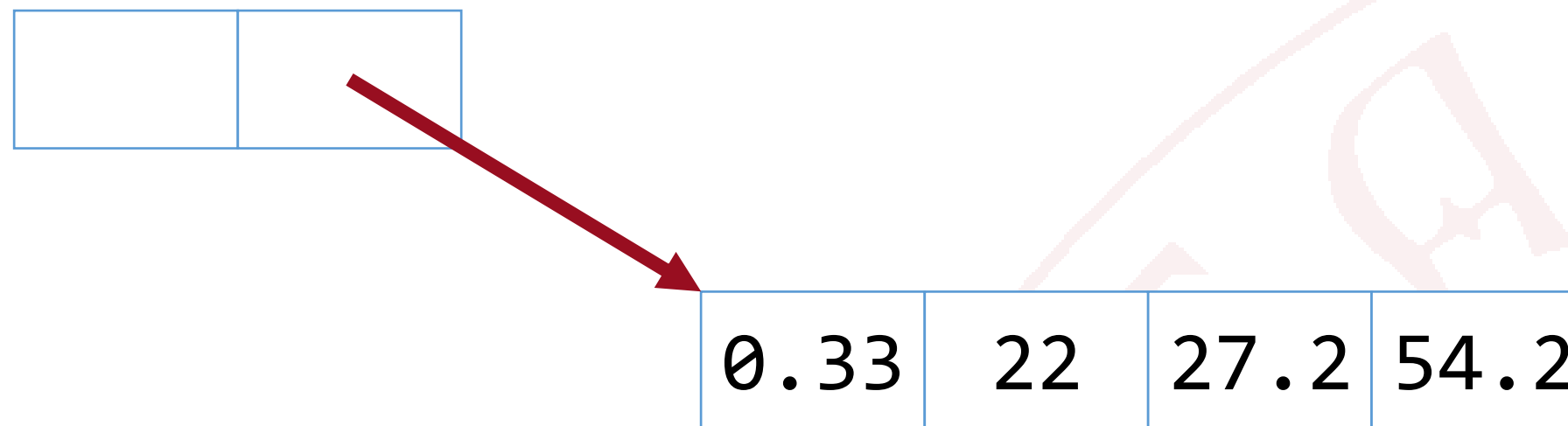
0.33	22	27.2	54.2
------	----	------	------

```
class vector {  
    int sz;  
    double* elem;  
  
    public:  
        vector(int s) : sz{ s }, elem{ new double[s] }  
        {  
            for (int i = 0; i < s; ++i) elem[i] = 0;  
        }  
    // ...  
};
```


Liberare la memoria con UDT

- Torniamo al nostro esempio di `vector<double>`

age:



- A fine vita, **è necessario deallocare la memoria**
- Come possiamo fare per essere *sicuri* che ciò avvenga in maniera *automatica*?
- Potrei usare una funzione `clean_up()`
 - Se l'utente si dimentica di chiamarla?

Distruttore

- Un distruttore è una funzione che il compilatore chiama quando un oggetto deve essere distrutto
 - Esce dallo scope
 - Deallocazione di un oggetto
- Il distruttore è chiamato **implicitamente**



Distruttore di vector

```
class vector {  
    int sz;  
    double* elem;  
  
public:  
    vector(int s) : sz{ s }, elem{ new double[s] }  
    {  
        for (int i = 0; i < s; ++i) elem[i] = 0;  
    }  
  
    ~vector()  
        { delete[] elem; }  
  
    // ...  
};
```

Initializzazione
degli elementi



Distruttore

Chiamata implicita del distruttore

```
void f3(int n){  
    double* p = new double[n];  
    vector v(n);  
    // ... uso p e v ...  
    delete[] p;  
}
```

v è liberato
automaticamente!

- vector fa automaticamente ciò che con p dobbiamo fare manualmente

Chiamata implicita del distruttore

- I distruttori sono molto utili quando dobbiamo rilasciare risorse acquisite
 - File
 - Thread
 - Lock
 - Stream
 - ...



Distruttori creati dal compilatore

- Quando il distruttore non è scritto esplicitamente, è generato automaticamente dal compilatore (**compiler-generated destructor**)
 - Non è "intelligente": **non dealloca al posto nostro**
 - Chiama i distruttori di eventuali oggetti membro

Distruttori creati dal compilatore

```
struct Customer {  
    std::string name;  
    std::vector<string> address;  
    // ...  
};  
  
void some_fct() {  
    Customer Fred;  
    // init Fred  
    // use Fred  
}
```

Qui Fred è distrutto:

- Chiamata a distruttore di `std::string`
- Chiamata a distruttore di `std::vector<string>`

Pattern acquisizione risorse

- Un oggetto acquisisce risorse nel costruttore
- Durante la sua vita, l'oggetto può:
 - Acquisire altre risorse
 - Liberare alcune risorse
- A fine vita, l'oggetto rilascia tutte le risorse



Azioni di new e delete

- Operatore **new**
 - Alloca memoria
 - Invoca costruttore

- Operatore **delete**
 - Invoca distruttore di vector
 - Libera memoria



Recap

- Deallocare la memoria
- Caccia al memory leak
- Passaggio di memoria dinamica tra funzioni e deallocazione nel chiamante
- Doppia cancellazione
- Distruttore

