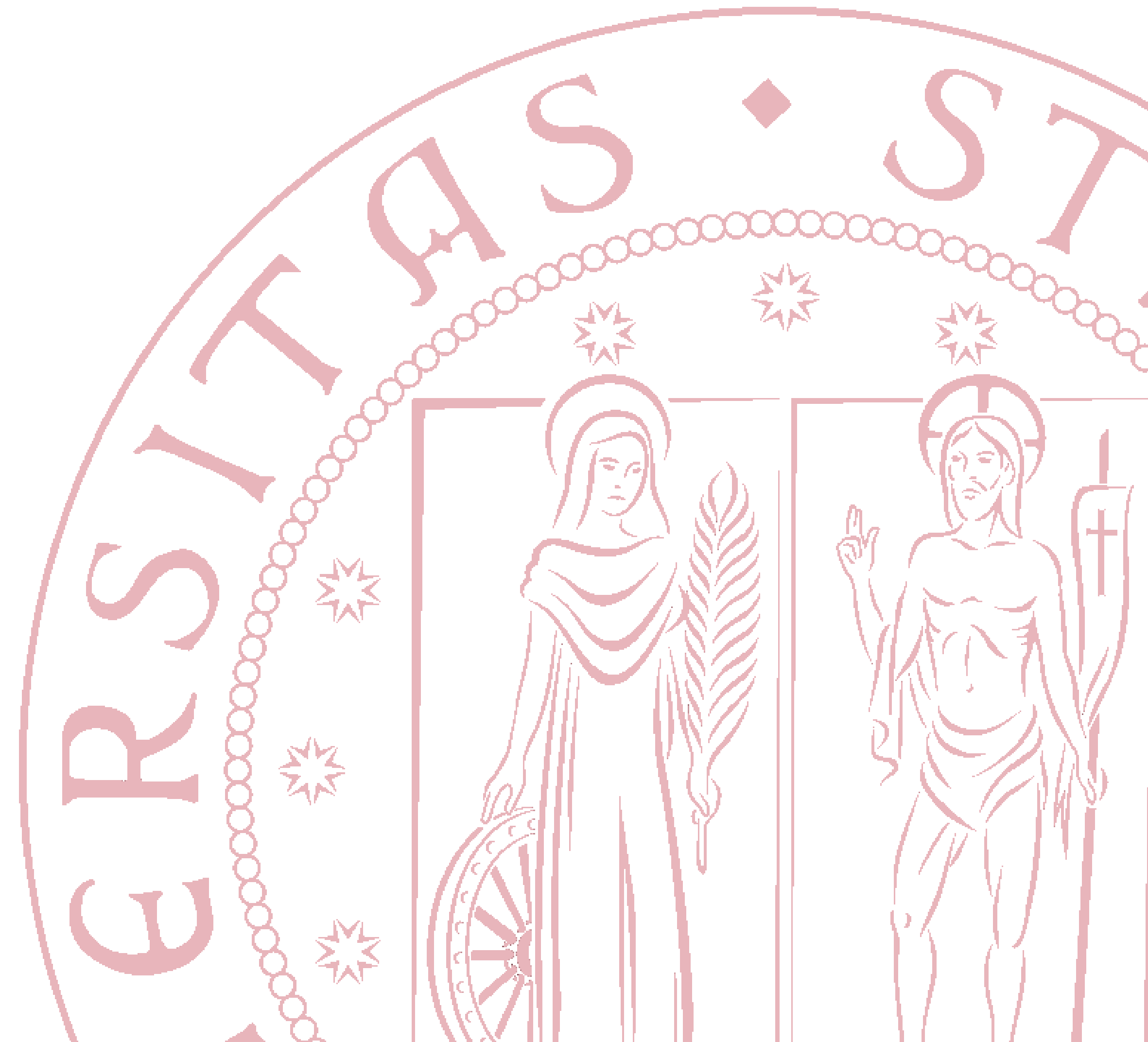


# 7.4 – Spostamento di oggetti

Libro di testo:

- Capitolo 18.3.4



# Agenda

- copy vs. move
- Move semantics



# Spostamento | Muovere i dati

```
vector fill(std::istream& is)
{
    vector res;
    for (double x; is >> x; ) res.push_back(x);
    return res;
}

void use()
{
    vector vec = fill(cin);
    // uso di vec
}
```

- **Quale effetto indesiderato è presente?**
  - Stiamo copiando una potenziale grande quantità di dati in res fuori da fill e poi dentro vec
- In realtà, difficilmente avremo mai un ulteriore bisogno di res all'interno di questa funzione

# Muovere i dati

- "We don't want a copy!" (BS)
- Non possiamo usare res dopo il return
  - Non possiamo usare contemporaneamente res e vec!
- Esiste un modo per spostare i dati, invalidando la sorgente?
  - "We would like to 'steal' the representation of res to use for vec." (BS)
- Ciò che cerchiamo è un'operazione di **move**
  - Complementa le operazioni di copia

# Move semantics

```
class vector {  
    int sz;  
    double* elem;  
  
public:  
    vector(vector&& a);           // move constructor  
    vector& operator=(vector&& a); // move assignment  
    // ...  
};
```

- La notazione **&&** è chiamata **rvalue reference**
- Gli argomenti non sono const (non possono esserlo!) – perché?
- Le definizioni di operazioni di move tendono a essere più semplici delle copie. Perché?

# Costruttore e assegnamento move

```
vector::vector(vector&& a) : sz{a.sz}, elem{a.elem} // ruba il puntatore ad a
{
    a.sz = 0; // annulla a
    a.elem = nullptr; // invalida il puntatore di a
}

vector& vector::operator=(vector&& a)
{
    delete[] elem; // dealloca lo spazio
    elem = a.elem; // riassegna il puntatore
    sz = a.sz; // riassegno il size
    a.elem = nullptr; // invalida il vecchio puntatore di a
    a.sz = 0; // azzera il size di a
    return *this; // ritorna una self-reference
}
```

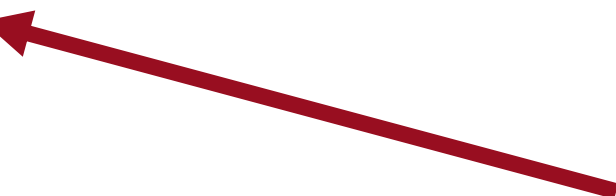
# fill() e use() con move constructor

- Riprendendo l'esempio iniziale

```
vector fill(istream& is)
{
    vector res;
    for (double x; is >> x; ) res.push_back(x);
    return res;
}

void use()
{
    vector vec = fill(cin);
    // uso di vec
}
```

**return implementato  
tramite move constructor**



- È il compilatore che implementa il return tramite move constructor

# Move constructor vs. copy elision

- In certe circostanze il move constructor non è chiamato
- Sostituito da una tecnica di ottimizzazione del compilatore – **copy elision**
  - L'oggetto è costruito direttamente nella funzione chiamante
- È possibile richiedere al compilatore di non applicare la copy elision
  - Opzione -fno-elide-constructors

<https://medium.com/swlh/c-rvalues-move-semantics-and-copy-elision-36d492da5446>

[https://en.cppreference.com/w/cpp/language/copy\\_elision](https://en.cppreference.com/w/cpp/language/copy_elision)



# Alternativa a move constructor

- Sappiamo che allocazione dinamica + puntatori permettono ai dati di uscire da una funzione

```
vector* fill2(istream& is)
{
    vector* res = new vector;
    for (double x; is >> x; ) res->push_back(x);
    return res;
}
void use2()
{
    vector* vec = fill2(cin);
    // ... usa vec ...
    delete vec;
}
```

- Questa versione, tuttavia, è molto più verbosa e soggetta a errori