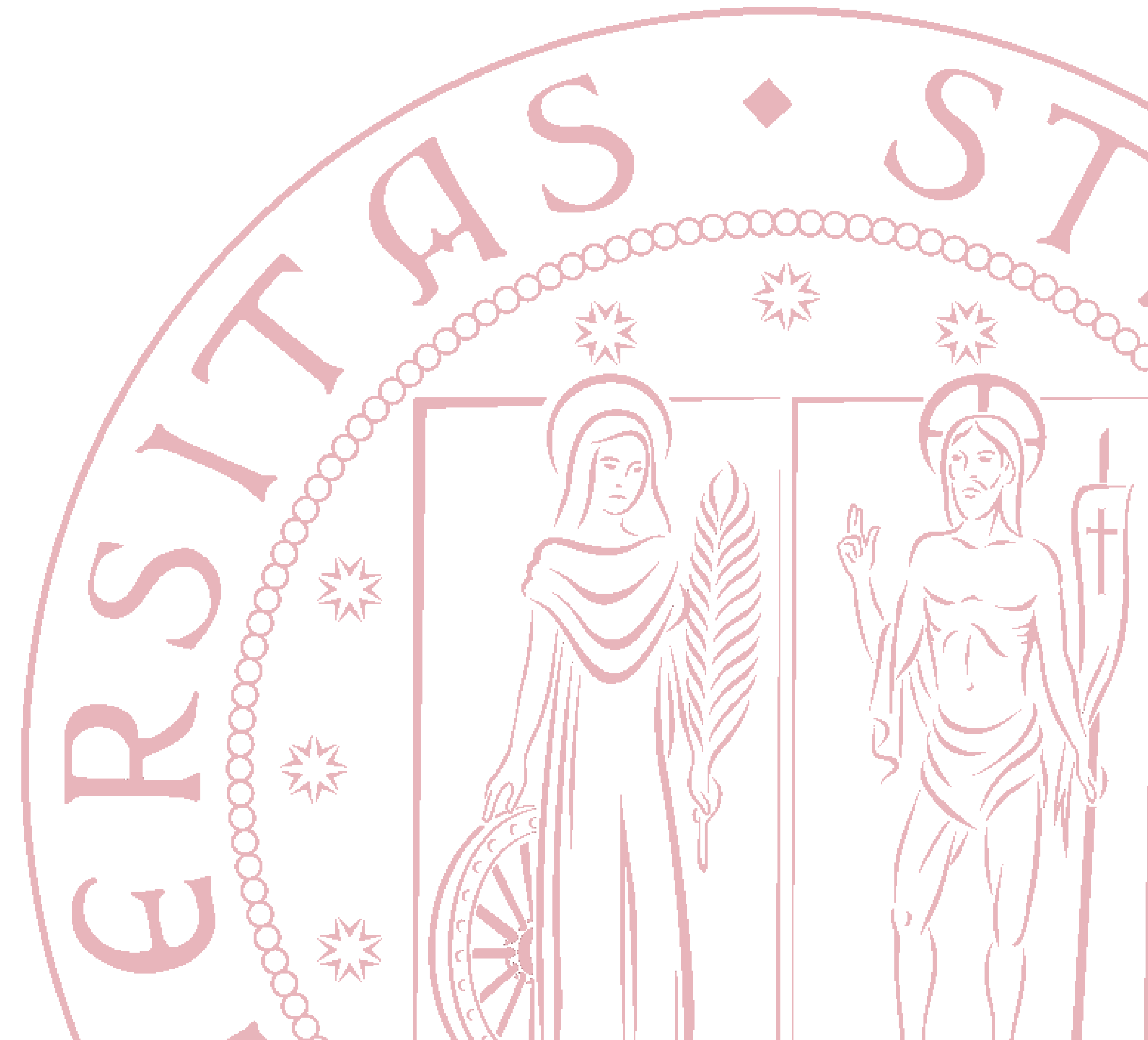


## 5.2 – Puntatori, reference, cast

Libro di testo:

Capitoli 17.8, 17.9, 17.9.1



# Agenda

- `void*`
- Cast
- Puntatori vs. reference



# void\* e cast (messing with types)



# void\*

- Abbiamo visto che i puntatori hanno un minimo check sul tipo
- **void\*** permette di saltare qualsiasi controllo
  - È un puntatore a memoria raw
  - Devo fornire indicazioni su come deve essere usata la memoria puntata
- Attenzione: void e void\*
- È possibile assegnare qualsiasi puntatore a un void\*
  - void\* rappresenta il concetto puro di "indirizzo di memoria" senza indicazioni su come usarla

# Esempio

```
int v1 = 7;
double v2 = 3.14;
void* pv1 = &v1;
void* pv2 = &v2;
```

Ok, ma perdo il  
type check!

```
void f(void* pv) {
    void* pv2 = pv;           // ok

    double* pd = pv;          // no! Conversione tra tipi
                                // incompatibili

    *pv = 7                    // no! non posso dereferenziare
                                // (che oggetto è?)

    pv[2] = 7;                 // no! Stessa ragione

    int* pi = static_cast<int*> (pv); // ok, conversione
                                    // esplicita
}
```

# Cast

- Conversione esplicita tra i tipi (inclusi i puntatori): **static\_cast**
  - **Check al tempo di compilazione, nessun check run-time**
  - **"A deliberately ugly name for an ugly and dangerous operation"**
- Due strumenti di conversione "**potentially even nastier**":

Nome cast	Effetto	
reinterpret_cast	Può fare il cast fra tipi totalmente indipendenti, es.: int e double*	Dici al compilatore: fidati di me, so quello che faccio! <b>MAI MAI MAI</b>
const_cast	Elimina const	

# Cast | I pochi casi in cui è giusto farlo

- I cast servono principalmente:
  - a interfacciarsi con HW
  - o altro codice non modificabile

Dice al compilatore che questa area di memoria deve essere interpretata come un Register

**OK!**

```
Register* in = reinterpret_cast<Register*>(0xff);  
  
void f(const Buffer* p){  
    Buffer* b = const_cast<Buffer*>(p);  
    // ...  
}
```

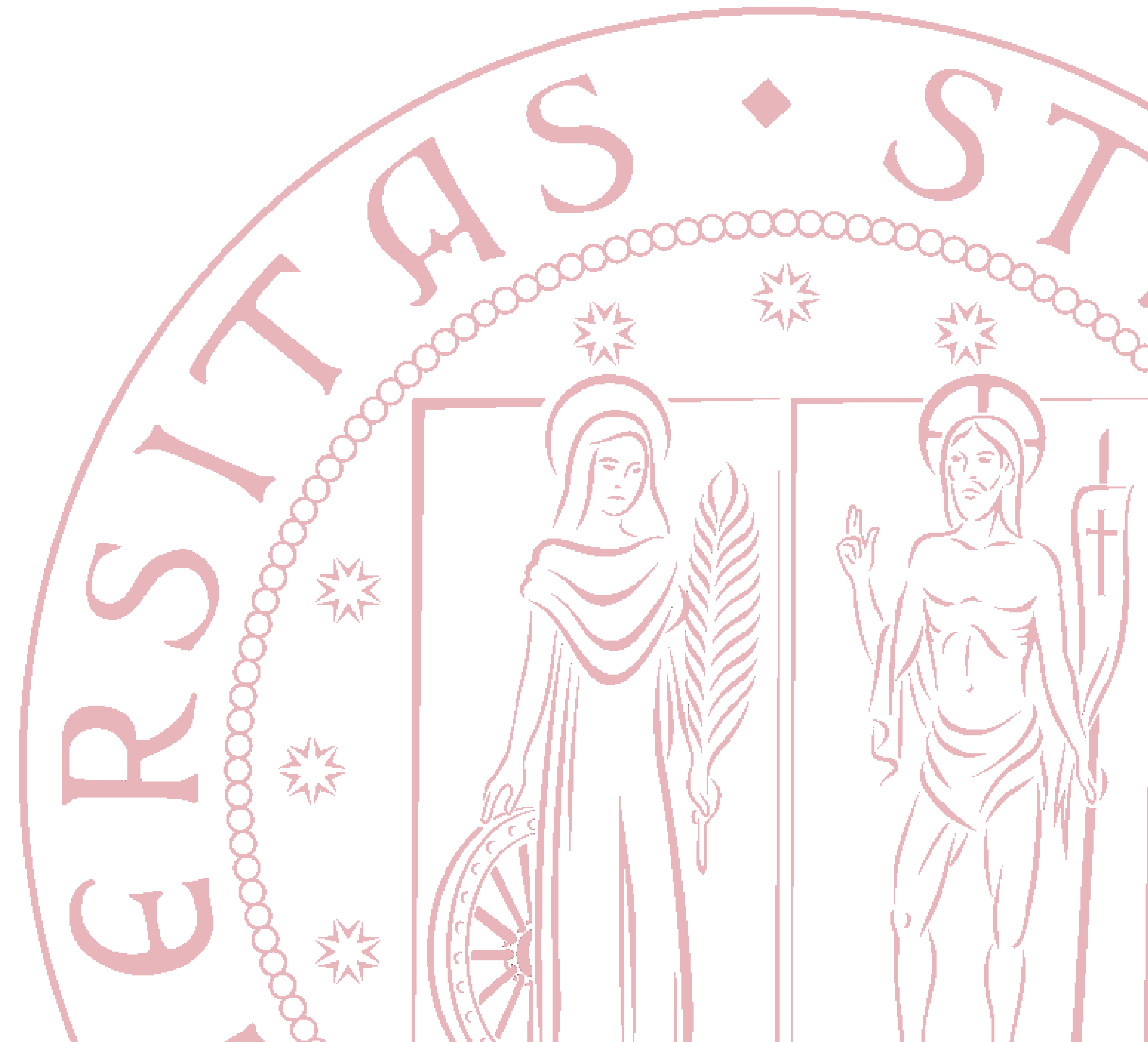
**OK!**

## Regole d'oro:

- Prima di utilizzare un cast, riguardare il codice e vedere se si può fare a meno
- Se si deve utilizzare, meglio lo `static_cast`

Necessario perché la libreria (di terze parti) che sto usando mi fornisce solo un `const Buffer*` che però io voglio modificare

# Puntatori e references





# Puntatori e reference

- Una reference è come un puntatore
  - immutabile
  - dereferenziato automaticamente
- ... oppure come un nome alternativo per un oggetto



# Puntatori vs. reference

- Puntatori

- Assegnamento: cambia il valore del puntatore, non dell'oggetto puntato
- Per cambiare il valore dell'oggetto puntato: **reference**

```
int x1 = 10;

int* p1 = &x1;

p1 = 7;      // sbagliato e
              // pericoloso!!

*p1 = 7;     // corretto
```

- Reference

- Assegnamento: cambio valore dell'oggetto

```
int y1 = 10;

int& r1 = y1;

r1 = 7;
```

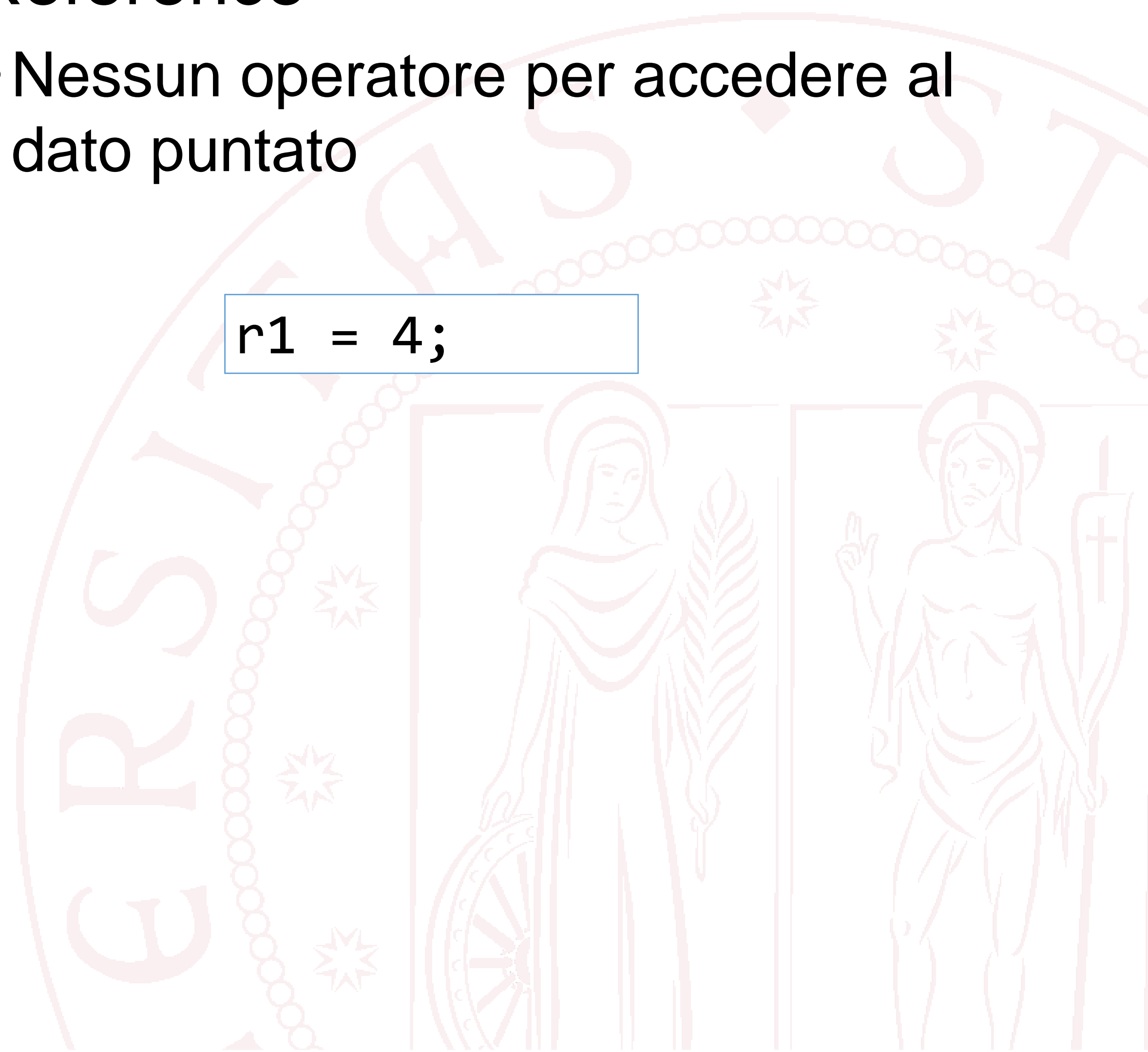
# Puntatori vs. reference

- Puntatori
  - Per accedere all'oggetto puntato: \* oppure [ ]

```
*p1 = 4;
```

- Reference
  - Nessun operatore per accedere al dato puntato

```
r1 = 4;
```



# Puntatori vs. reference

- Puntatori

- Assegnamento: copia del puntatore, non dell'oggetto puntato (**shallow copy**)

```
int i, j;  
int *p1 = &i;  
int *p2 = &j;  
p2 = p1;           // copia del  
                    // puntatore
```

- Reference

- Assegnamento: copia dell'oggetto a cui si riferisce (**deep copy**)

```
int i, j;  
int &r1 = i;  
int &r2 = j;  
r2 = r1;           // copia del  
                    // contenuto
```

- **Deep copy** vs. **shallow copy**: implicazioni nell'uso della memoria con gli UDT

# Puntatori vs. reference

- Puntatori
  - Esiste il null pointer

```
int *p1 = nullptr;
```

- Reference
  - Non esiste una reference non valida



# Parametri puntatori e reference

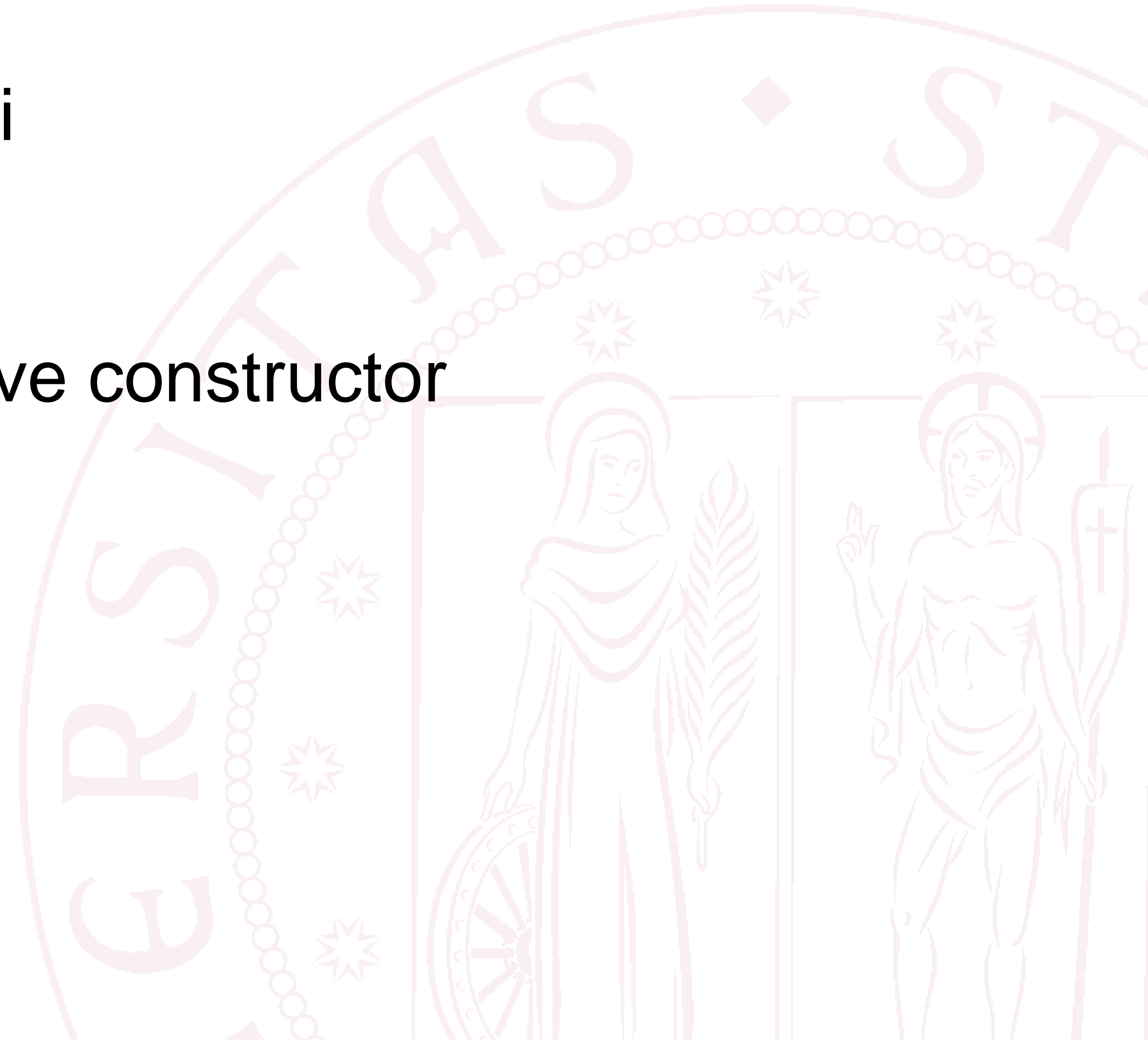
- Chiamata a funzione: abbiamo visto gli effetti di una modifica di parametri reference
- Con i puntatori si può ottenere lo stesso effetto
- Quindi, volendo cambiare il valore di una variabile ho tre opzioni:

```
int incr_v(int x) { return x + 1; }  
  
void incr_p(int* p) { ++*p; }  
  
void incr_r(int& r) { ++r; }
```

- Come scegliere?

# Parametri puntatori e reference

- Ritornare il valore:
  - È più chiaro e meno soggetto a errori
  - Ok per oggetti piccoli
  - Ok per oggetti grandi se hanno il move constructor



# Parametri puntatori e reference

- Reference vs. puntatore:
  - I puntatori sono espliciti

```
int incr_v(int x) { return x + 1; }  
  
void incr_p(int* p) { ++*p; }  
  
void incr_r(int& r) { ++r; }
```

```
int x = 7;
```

```
incr_p(&x);
```

Esplicito

```
incr_r(x);
```

"Looks innocent"



# Parametri puntatori e reference

- Reference vs. puntatore:
  - I puntatori possono aver valore `nullptr`, le reference no

```
int* p = nullptr;

incr_p(p);           // incr_p deve gestire questo caso

void incr_p(int* p) {
    if (p == nullptr) error("null pointer argument");
    ++*p;
}
```

# Parametri puntatori e reference

- Reference vs puntatore – un criterio:
  - Se no-object è un valore plausibile: puntatore
  - Altrimenti: reference/const reference



# Recap

- Puntatori a void: gestire la memoria grezza
- Cast
- Puntatori vs reference
  - Differenze sintattiche
  - Differenze espressive
- Deep copy vs shallow copy

