# 6 – Liste

Libro di testo:

Capitoli 17.9.3-5, 17.10

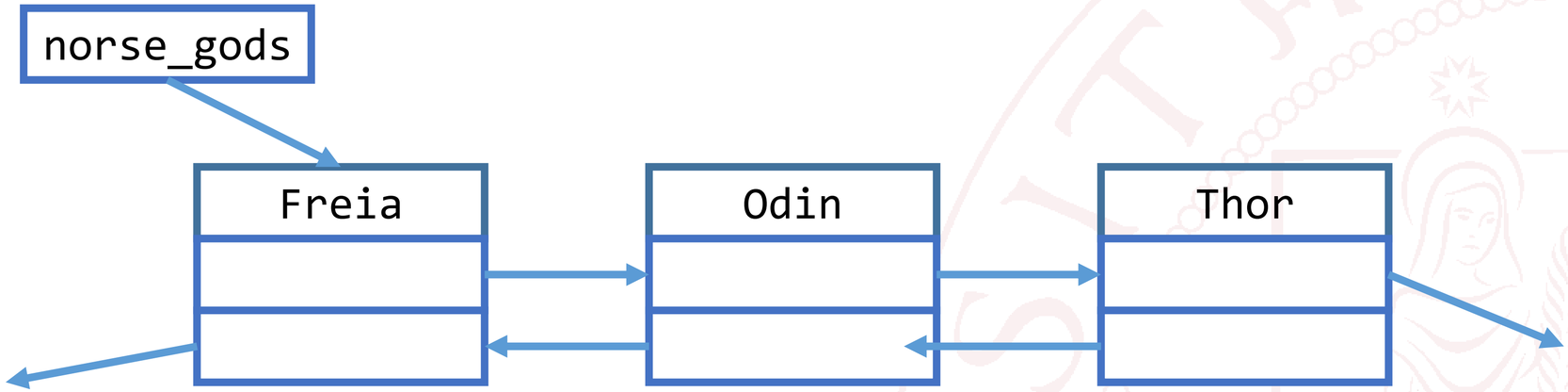
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

# Agenda

- Lista come struttura dati
- Implementazione di una lista tramite puntatori
- Puntatore `this`

# Lista

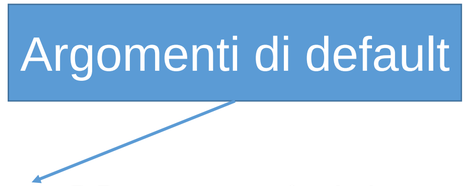
- Struttura dati con caratteristiche specifiche
  - Accesso seriale
  - Le operazioni di aggiunta e rimozione di un nodo sono veloci
  - Utilizzo dell'allocazione dinamica




- Una lista è fatta di «link» che possiedono informazioni sul dato e puntatori ad altri link
- Una lista double-linked ha puntatori ai link precedente e successivo

# Lista

```
struct Link {
    std::string value;
    Link* prev;
    Link* succ;
    Link(const std::string& v, Link* p = nullptr, Link* s = nullptr)
    : value{v}, prev{p}, succ{s} {}
};
```

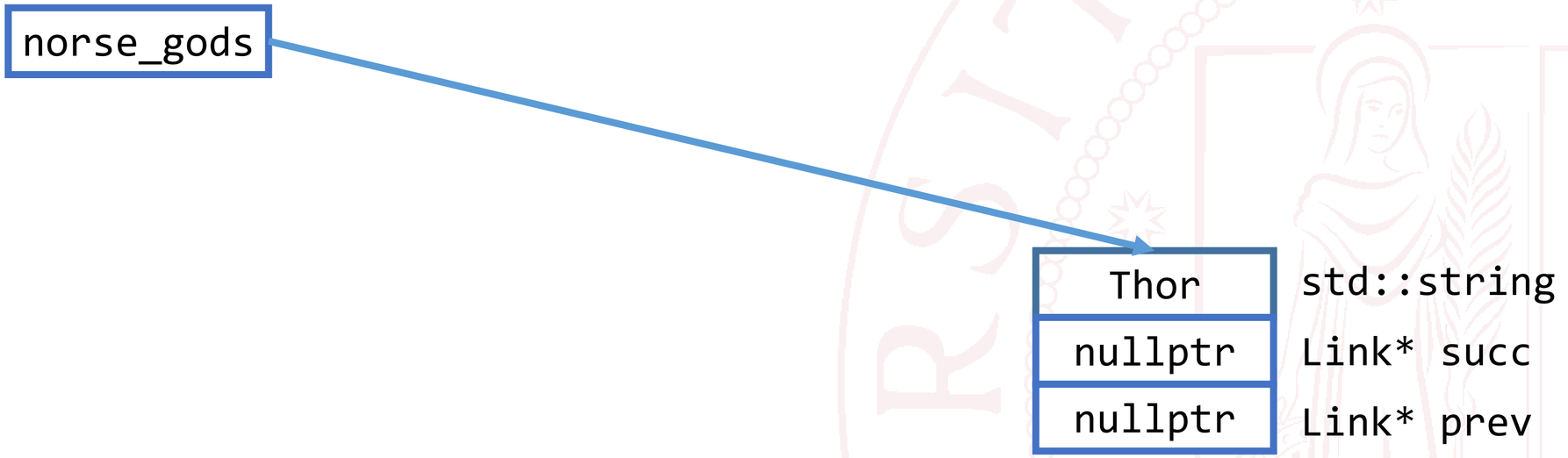



- Per accedere a un membro di `Link`, useremo l'operatore →
  - Equivale a **`(*…).xxxx`**
  - `(*p).field` equivale a `p->field`

# Esempio

Per costruire la lista:

```
Link* norse_gods = new Link{"Thor", nullptr, nullptr};
```

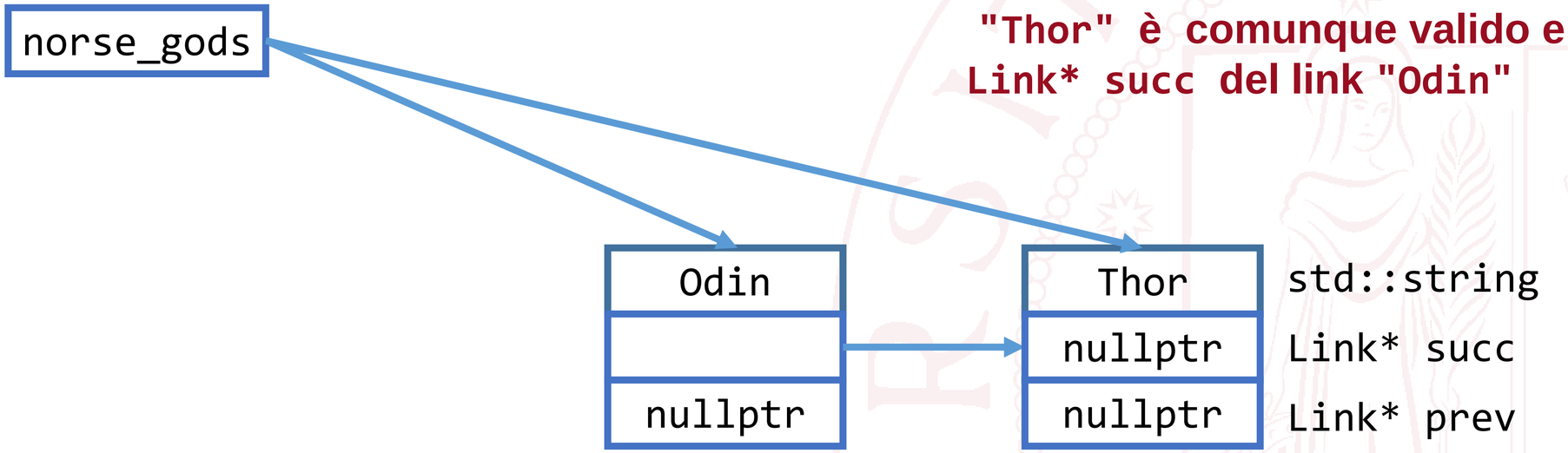


# Esempio

Per costruire la lista:

```
Link* norse_gods = new Link{"Thor", nullptr, nullptr};
norse_gods = new Link{"Odin", nullptr, norse_gods};
```

**Notare che `norse_gods` è sovrascritto ma l'indirizzo all'area di memoria del link `"Thor"` è comunque valido e salvato in `Link* succ` del link `"Odin"`**

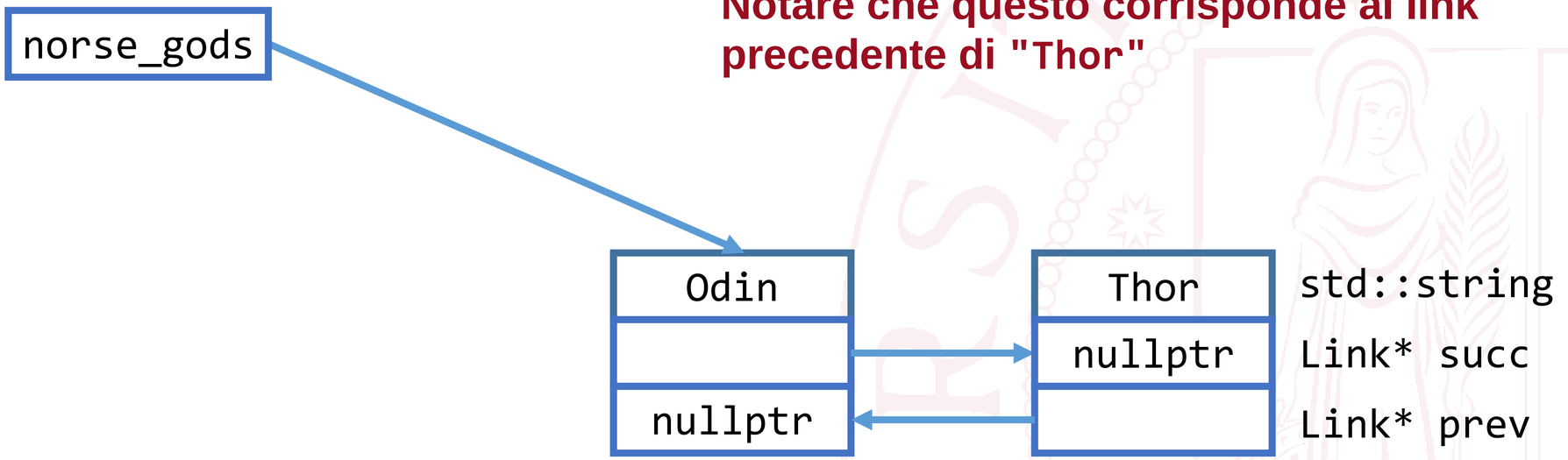


# Esempio

Per costruire la lista:

```
Link* norse_gods = new Link{"Thor", nullptr, nullptr};
norse_gods = new Link{"Odin", nullptr, norse_gods};
norse_gods->succ->prev = norse_gods;
```

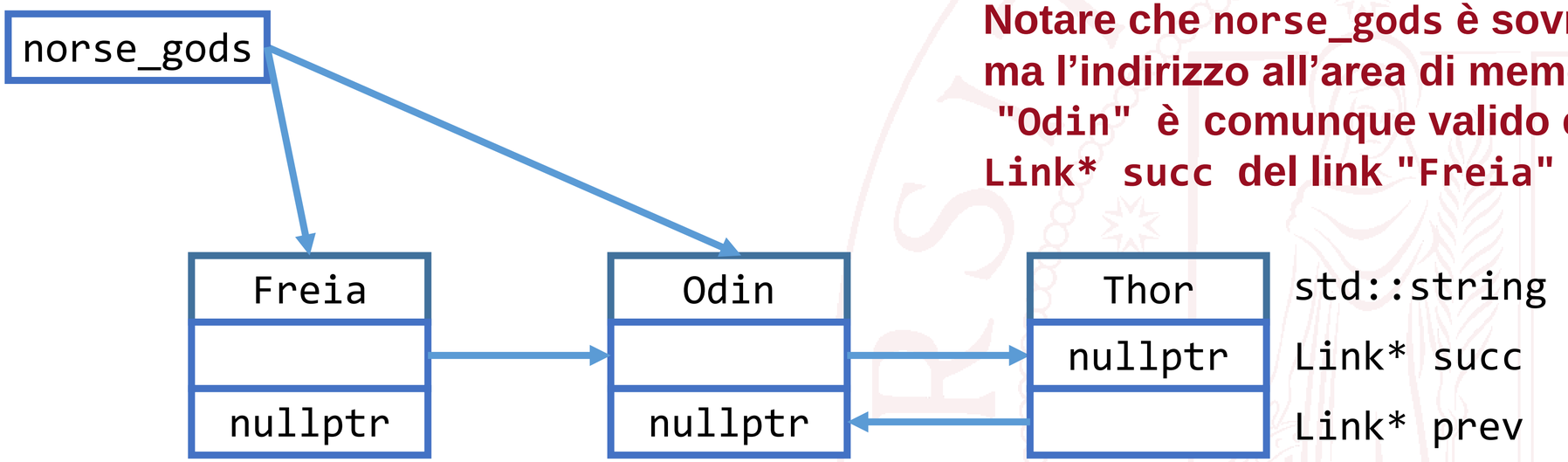
**Notare che questo corrisponde al link precedente di "Thor"**



# Esempio

Per costruire la lista:

```
Link* norse_gods = new Link{"Thor", nullptr, nullptr};
norse_gods = new Link{"Odin", nullptr, norse_gods};
norse_gods->succ->prev = norse_gods;
norse_gods = new Link{"Freia", null_ptr, norse_gods};
```

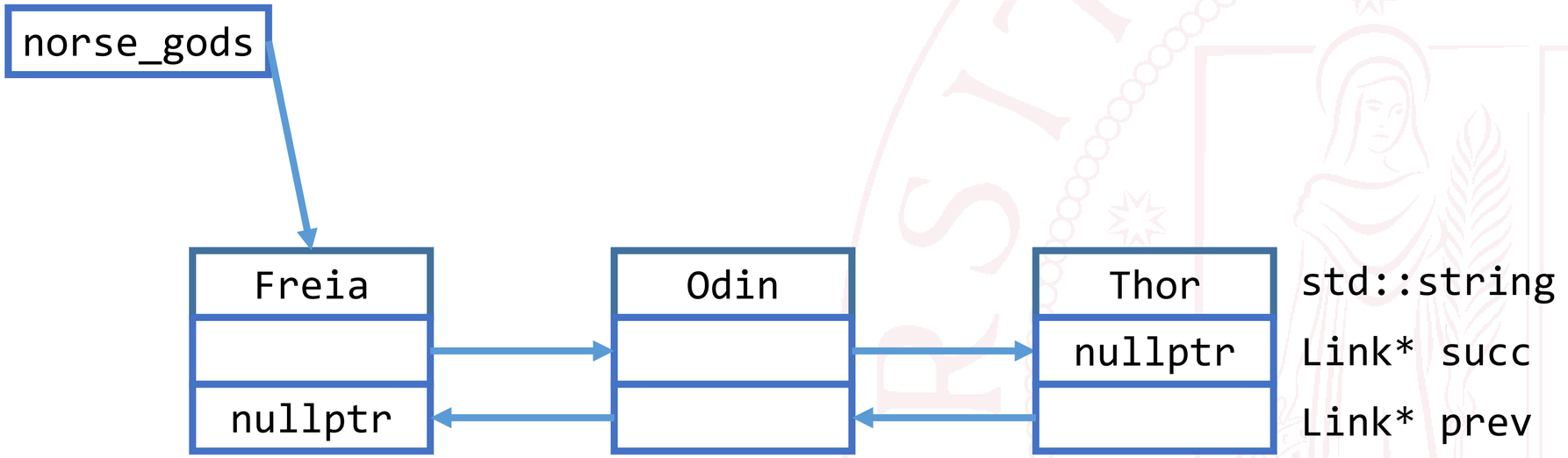



**Notare che `norse_gods` è sovrascritto ma l'indirizzo all'area di memoria del link `"Odin"` è comunque valido e salvato in `Link* succ` del link `"Freia"`**

# Esempio

Per costruire la lista:

```
Link* norse_gods = new Link{"Thor", nullptr, nullptr};
norse_gods = new Link{"Odin", nullptr, norse_gods};
norse_gods->succ->prev = norse_gods;
norse_gods = new Link{"Freia", null_ptr, norse_gods};
norse_gods->succ->prev = norse_gods;
```



# Funzione `insert`

- L'inserimento è un'operazione molto comune!
  - Ha senso usare una funzione dedicata

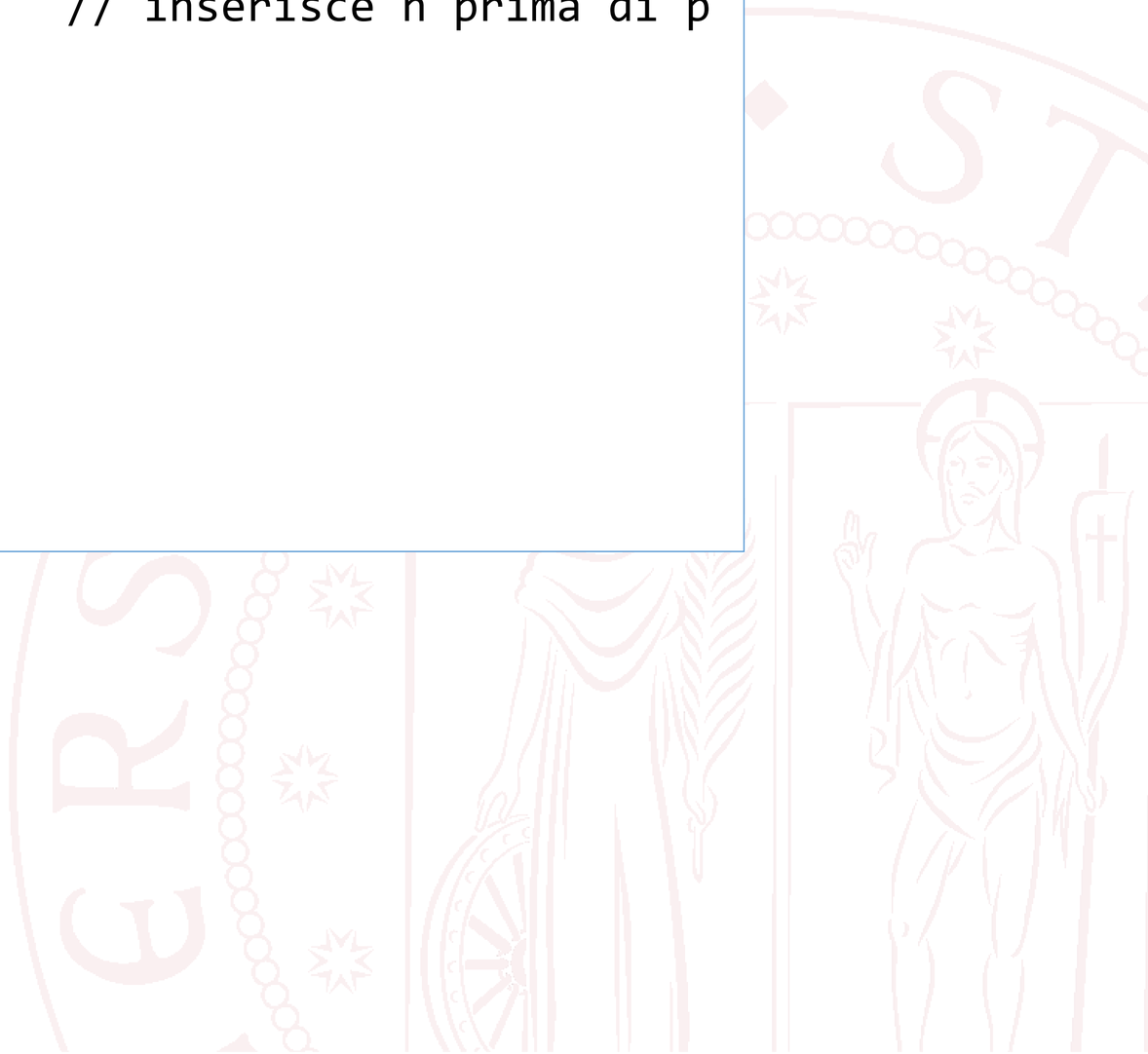
```
Link* insert(Link* p, Link* n) {     // inserisce n prima di p
    n->succ = p;
    p->prev->succ = n;
    n->prev = p->prev;
    p->prev = n;
    return n;
}
```

  - Vincolo: questa funziona se `p` punta a un `Link`, e se questo ha un `prev`!

# Funzione `insert` con check per `nullptr`

```
Link* insert(Link* p, Link* n) {      // inserisce n prima di p
    if (!n) return p;
    if (!p) return n;

    n->succ = p;
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev;
    p->prev = n;
    return n;
}
```

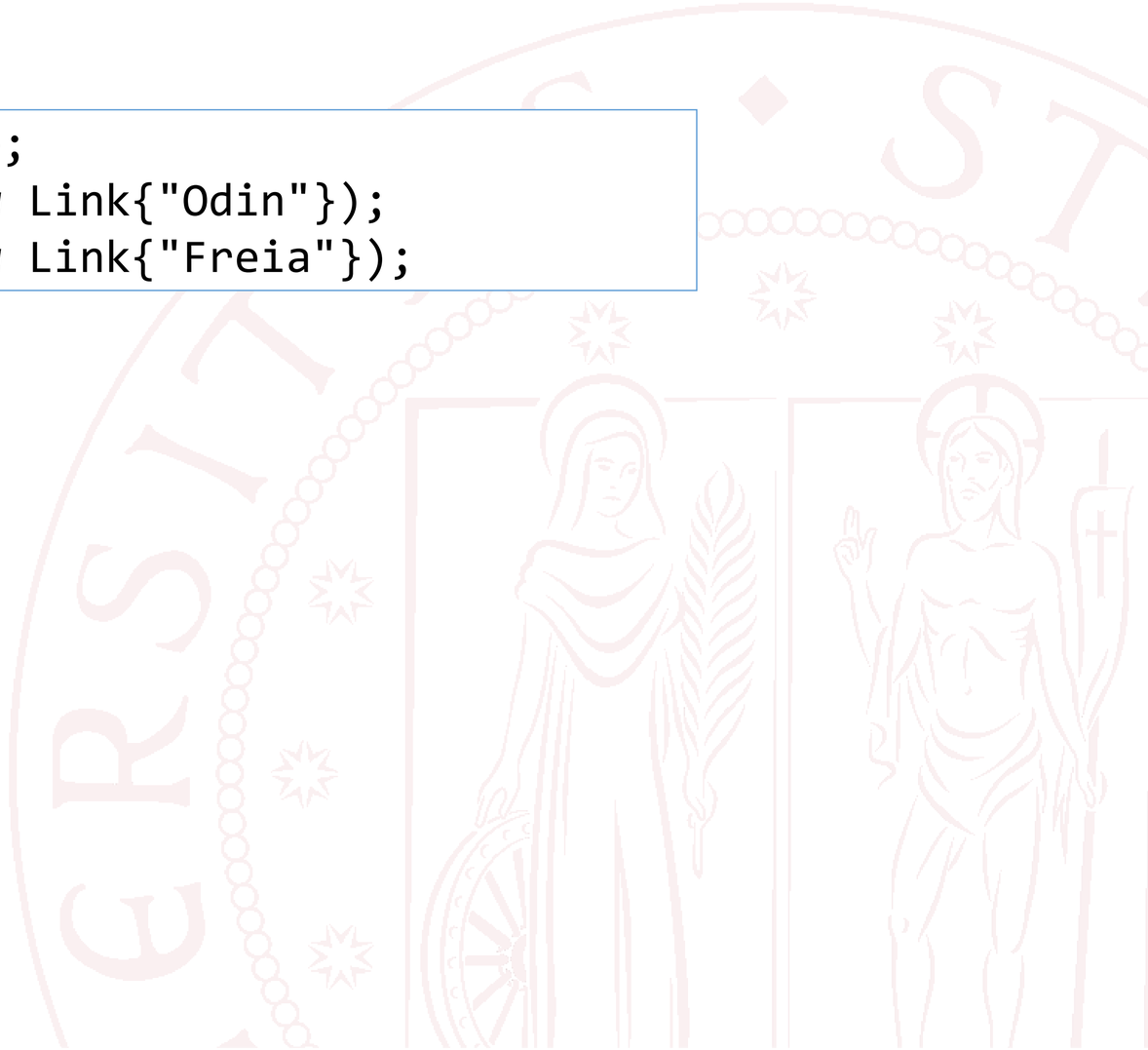


# Usare la funzione insert

- Ora possiamo inserire i link più facilmente

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods, new Link{"Odin"});
norse_gods = insert(norse_gods, new Link{"Freia"});
```

- La gestione dei puntatori è nascosta!



# Operazioni su liste

- Funzioni utili per la gestione di una lista:
  - **Costruttore**
  - **`insert`:** inserimento prima di un elemento
  - **`add`:** inserimento dopo un elemento
  - **`erase`:** rimozione di un elemento
  - **`find`:** trova un `Link` con un certo valore
  - **`advance`:** trova l'n-simo successivo

# Operazioni su liste | Esercizio

```
Link* add(Link* p, Link* n) {...} // esercizio
Link* erase(Link* p)
{
    if(!p) return nullptr;
    if(p->succ) p->succ->prev = p->prev;
    if(p->prev) p->prev->succ = p->succ;

    return p->succ;   // attenzione: ritorna un puntatore ma
                      // non dealloca!
}
```

# Operazioni su liste

```
Link* find(Link* p, const string& s)
{
    while(p)
    {
        if(p->value == s) return p;
        p = p->succ;                    // anziché ++p
    }
    return nullptr;
}
```

# Operazioni su liste

- Che operazione esegue questa funzione?

```
Link* advance(Link* p, int n) {

    if(!p) return nullptr;

    if(0 < n) {
        while(n--) {
            if(!p->succ) return nullptr;
            p = p->succ;
        }
    } else if(n < 0) {
        while(n++) {
            if(!p->prev) return nullptr;
            p = p->prev;
        }
    }
    return p;
}
```

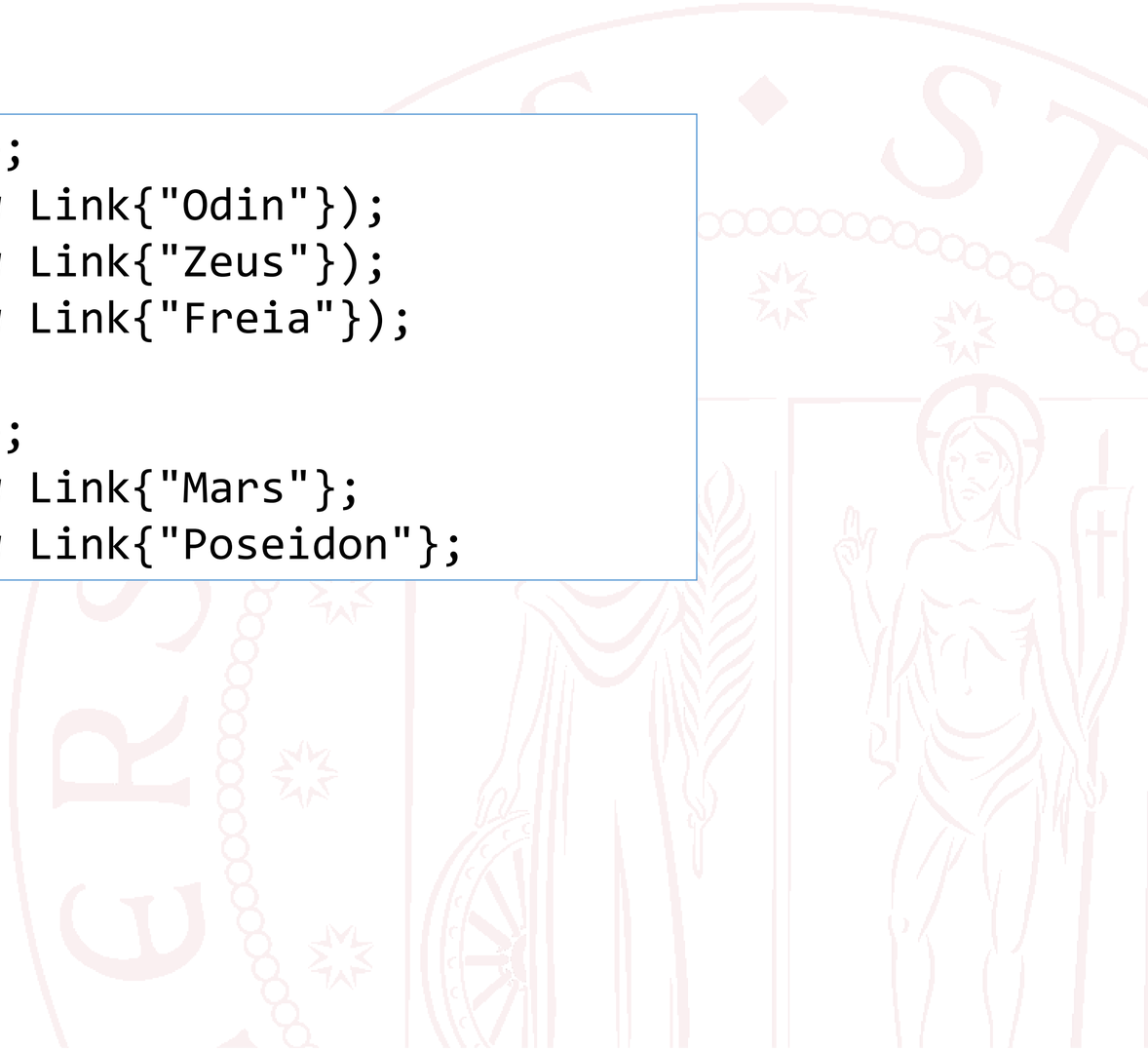
# Utilizzo delle liste

- Ora posso usare le liste così:

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods, new Link{"Odin"});
norse_gods = insert(norse_gods, new Link{"Zeus"});
norse_gods = insert(norse_gods, new Link{"Freia"});

Link* greek_gods = new Link{"Hera"};
greek_gods = insert(greek_gods, new Link{"Mars"};
greek_gods = insert(greek_gods, new Link{"Poseidon"};
```

- Ma: Zeus è greco, e Marte è romano
  - Come modificare le liste?



# Modifica di liste

```
Link* p = find(norse_gods, "Zeus");

if(p)
{
    erase(p);
    insert(greek_gods, p);
}
```

- Funziona, ma ha due problemi:
  - Cosa succede se p è norse_gods?
  - greek_gods  non è aggiornato

# Modifica di liste

```
Link* p = find(norse_gods, "Zeus");

if(p)
{
    if (p == norse_gods) norse_gods = p->succ;
    erase(p);
    greek_gods = insert(greek_gods, p);
}
```

# Stampa di liste

```
void print_all(Link* p)
{
    cout << "{";
    while(p)
    {
        cout << p->value;
        if(p = p->succ) cout << ", ";
    }
    cout << "]";
}

print_all(norse_gods);
cout << "\n";
print_all(greek_gods);
```

# Puntatore `this`

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

# Migrazione a funzioni membro?

- Molte delle funzioni viste prima accettano un `Link*` come argomento
  - Ha senso che diventino funzioni membro?
  - Sì, e ciò nasconde all'esterno l'uso dei puntatori
- Nuova versione della classe

# Classe Link

```
class Link {
  public:
      string value;                    // public: sono solo dati

      Link(const string& v, Link* p = nullptr, Link* n =   nullptr)
    : value{v}, prev{p}, succ{s} {}

      Link* insert(Link* n);           // inserimento prima di questo
      Link* add(Link* n);              // inserimento dopo questo
      Link* erase();                   // rimuove questo
      Link* find(const string& s);
      const Link* find(const string& s) const;

      Link* advance(int n) const;

      Link* next() const { return succ; }
      Link* previous() const { return prev; }
  private:
      Link* prev;
      Link* succ;
};
```

# Inserimento prima di questo oggetto

- Se devo inserire prima di *questo* oggetto, ho bisogno di un riferimento
  - **`this`** è un puntatore a questo oggetto

```
Link* Link::insert(Link* n)
{
    Link* p = this;                 // posso anche lasciare this
    if (n == nullptr) return p;
    n->succ = p;
    if(p->prev) p->prev->succ = n;
    n->prev = p->prev;
    p->prev = n;

    return n;
}
```

# Puntatore this

- `this` è immutabile

```
struct S {
    // ...
    void mutate (s* p)
    {
        this = p;              // errore!
        // ...
    }
};
```