10.2 – Standard Template Library (STL)

Linked list e il suo iteratore

Libro di testo:

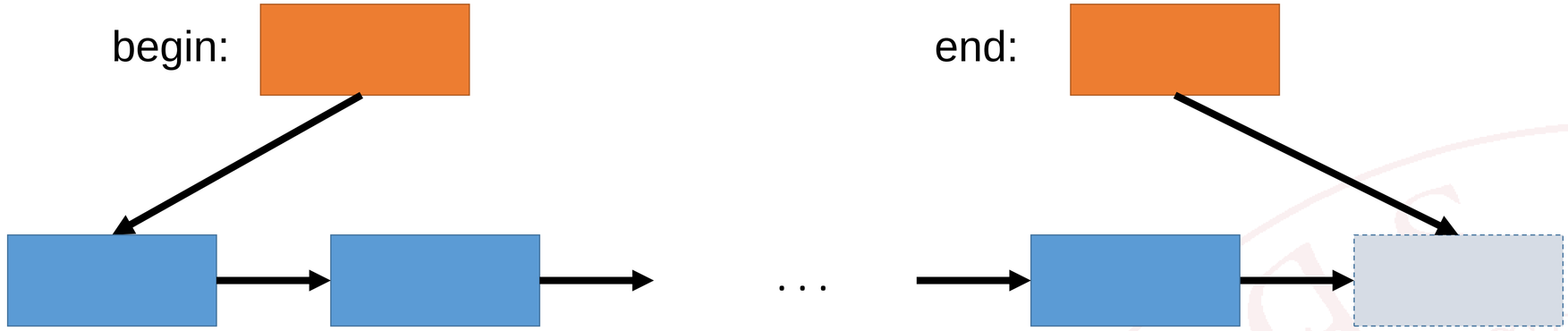
- Capitoli 20.4, 20.5

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

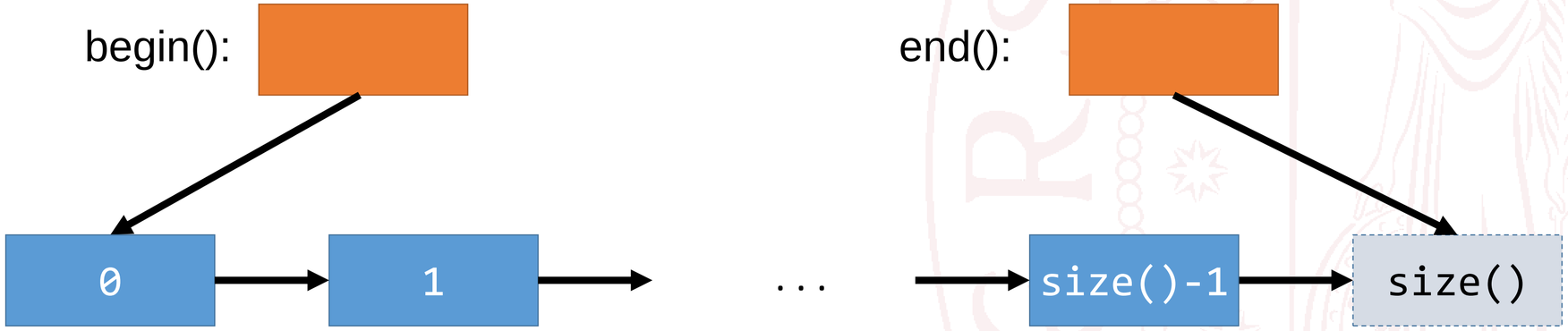
Agenda

- Esempio di implementazione (molto parziale) di linked list STL
- Implementazione degli iteratori
- `auto`

std::vector come sequenza

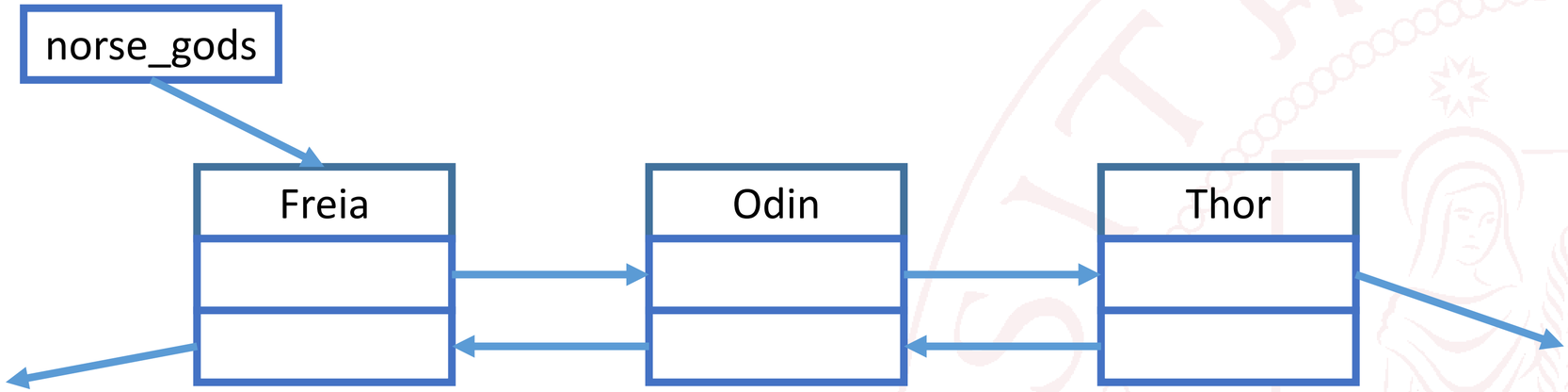


- Nel caso di `std::vector`, gli elementi sono consecutivi in memoria
- Dall'element `0` all'elemento `size-1`



Un esempio: linked list

- La nozione STL di sequenza non prevede che gli elementi siano consecutivi
- La struttura dati più vicina a una sequenza STL è:



Linked list

- Progettiamo alcune parti di una linked list
  - Strutture coinvolte:

```
template<typename Elem>
struct Link {
    Link* prev;                         // Link precedente
    Link* succ;                         // Link successivo
    Elem val;                           // Valore
};

template<typename Elem>
struct list {
    Link<Elem>* first;
    Link<Elem>* last;
};
```

Linked list – operazioni

- Dobbiamo progettare le operazioni che sarà possibile effettuare su una `list`
  - Quelle di `vector` (senza `[]`) – costruttore, size, ...
  - `insert` ed `erase`
  - Iteratori per attraversare la lista
- In STL, un iteratore è un membro della classe container
- Perché è stato escluso `[]`?

Linked list

```
template<typename Elem>
class list {
    // rappresentazione e dettagli implementativi
    public:
      class iterator;

      iterator begin();
      iterator end();

      iterator insert(iterator p, const Elem& v);
      iterator erase(iterator p);

      void push_back(const Elem& v);
      void push_front(const Elem& v);
      void pop_front();
      void pop_back();

      Elem& front();
      Elem& back();
      // ...
};
```

Cos'è questo?
È corretto?

Restituiscono un iteratore!

Iteratore di lista

- Dopo aver progettato `list`, dobbiamo progettare il suo iteratore
  - Deve contenere un puntatore a un elemento della lista
  - Deve implementare i seguenti operatori:
    - `++`, `--`, `*`, `==`, `!=`

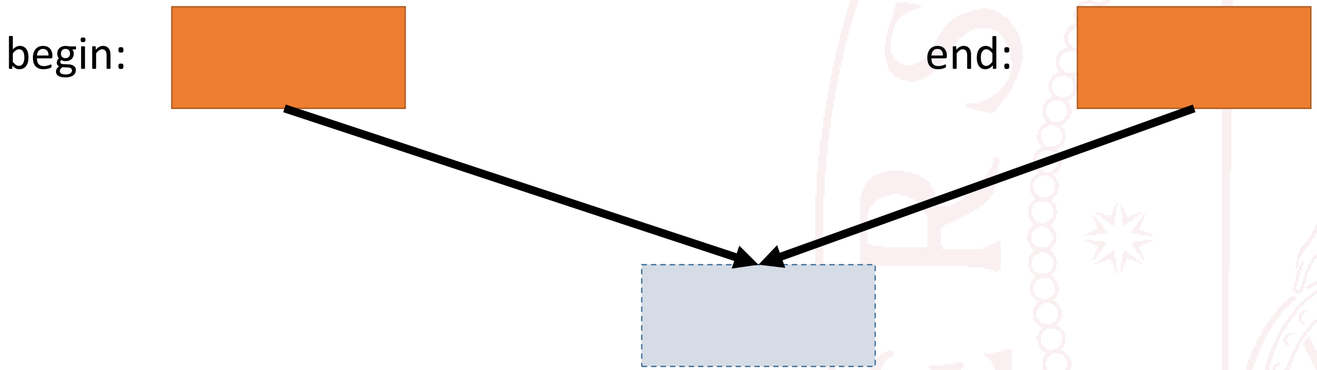
Iteratore di lista

```
template<typename Elem>
class list<Elem>::iterator {
    Link<Elem>* curr;
    public:
        iterator(Link<Elem>* p) : curr{p} {}

        iterator& operator++() { curr = curr->succ; return *this; }

        iterator& operator--() { curr = curr->prev; return *this; }

        Elem& operator*() { return curr->val; }

        bool operator==(const iterator& b) const { return curr == b.curr; }

        bool operator!=(const iterator& b) const { return curr != b.curr; }
};
```

Liste vuote

- Se la lista è vuota, `begin == end`
  - È la condizione da verificare

- **end che punta a un elemento dopo la fine permette di trattare la lista vuota come un caso *non* speciale**



Iteratori e relativo codice

- Gli iteratori tendono a rendere il codice complesso

```
template<typename T>
void user(vector<T>& v, list<T>& lst)
{
    for (vector<T>::iterator p = v.begin(); p != v.end(); ++p) {
      cout << *p << '\n';
    }

    list<T>::iterator q = find(lst.begin(), lst.end(), T{42});
}
```

Iteratori e relativo codice

- Molto spesso il tipo dell'iteratore può essere dedotto dal compilatore:

```
std::vector<T>::iterator p = v.begin();
```

- Dato che `p` è inizializzato con `v.begin()`, non può che essere un `std::vector<T>::iterator`

auto

- auto permette di semplificare questa notazione

```
template<typename T>
void user(vector<T>& v, list<T>& lst)
{
    for (auto p = v.begin(); p != v.end(); ++p) {
       cout << *p << '\n';
    }

    auto q = find(lst.begin(), lst.end(); T{42});
}
```

auto

- `auto` ha un utilizzo più generale:

```
auto x = 123;      // x è un int
auto c = 'y';      // c è un char
auto& r = x;       // r è un int&
auto y = r;        // y è un int (le reference sono
                   // implicitamente dereferenziate)

// attenzione!
auto s1 = "San Antonio";                  // s1 è const char*
std::string s2 = "Fredericksburg";        // s2 è una string
```

- `auto` può portare ad ambiguità
  - Da usare con attenzione (consigliato solo per gli iteratori, **sconsigliato** in tutti gli altri casi!)

Recap

- Implementazione di list (a partire da link)
- Implementazione dell'iteratore di list
- Uso di `auto`