7.2 – Copia di oggetti

Libro di testo:

- Capitolo 18.3

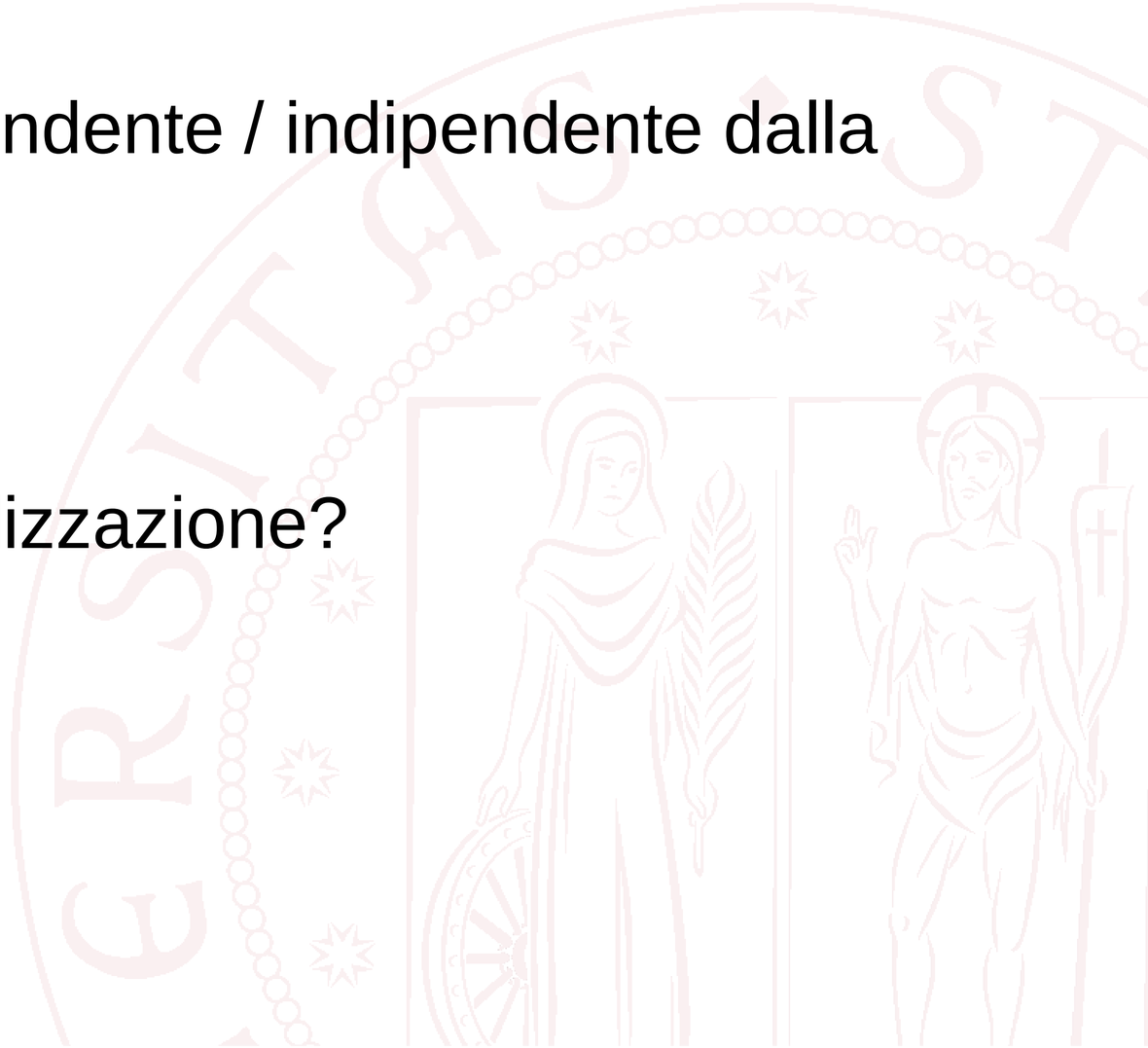
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Copia
  - shallow vs deep copy
- Costruttore di copia
- Assegnamento di copia

Copia

- Cosa significa copiare un oggetto?
- In che modo un oggetto copiato è dipendente / indipendente dalla sorgente della copia?
- Che operazioni di copia esistono?
- In che modo la copia è legata all'inizializzazione?



Recap (come prima)

- Riprendiamo il nostro `vector`

```
class vector {
    int sz;
    double* elem;

    public:
        vector(int s)
            : sz{s}, elem{new double[s]} { /* ... */ }
        ~vector()
            { delete[] elem; }
        // ...
};
```

Copia

- Cosa succede se copio un `vector`?

```
void f(int n)
{




}
```

Copia

- Cosa succede se copio un `vector`?

```
void f(int n)
{
    vector v(3);      // definisce un vettore di 3 elementi



}
```

Copia

- Cosa succede se copio un `vector`?

```
void f(int n)
{
    vector v(3);       // definisce un vettore di 3 elementi
    v.set(2, 2.2);     // assegna a v[2] il valore di 2.2


}
```

Copia

- Cosa succede se copio un `vector`?

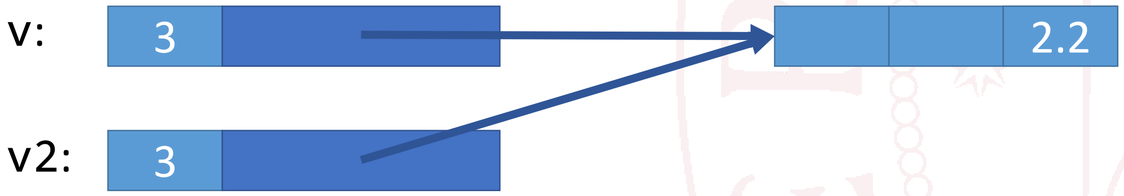
```
void f(int n)
{
    vector v(3);        // definisce un vettore di 3 elementi
    v.set(2, 2.2);      // assegna a v[2] il valore di 2.2
    vector v2 = v;      // cosa succede qui?
    // ...
}
```

Copia

- Cosa succede se copio un `vector`?

```
void f(int n)
{
    vector v(3);        // definisce un vettore di 3 elementi
    v.set(2, 2.2);      // assegna a v[2] il valore di 2.2
    vector v2 = v;      // cosa succede qui?
    // ...
}
```

- Default: copia membro a membro

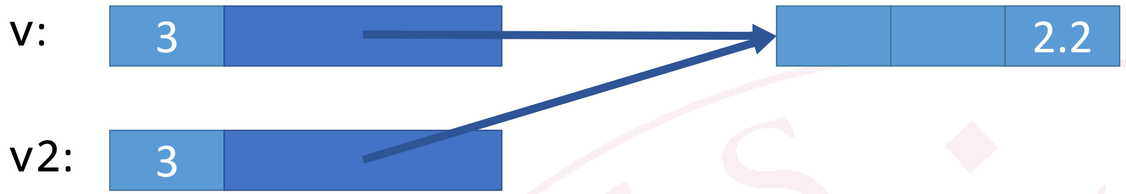



```
// Recall
class vector {
    int sz;
    double* elem;
    // ...
};
```

```
v.sz   == v2.sz
v.elem == v2.elem
```

Copia

```
void f(int n)
{
    vector v(3);
    v.set(2, 2.2);
    vector v2 = v;
}
```



- v2 <u>non ha una copia</u> degli elementi:
  - condivide gli stessi elementi di v – **shallow copy**
- Crea problemi?
- In uscita da f, sono chiamati i distruttori di v e v2
  - È un problema?

Una volta chiamato il distruttore, la memoria puntatata da `elem`, viene liberata due volte → errore!

Costruttore di copia

- Vogliamo modificare il comportamento della copia di default
- Dobbiamo definire il costruttore di copia
  - Costruttore che accetta una reference a oggetto della stessa classe

```
class vector {
    int sz;
    double* elem;

    public:
        vector(const vector&);   // costruttore di copia
        // ...
};
```

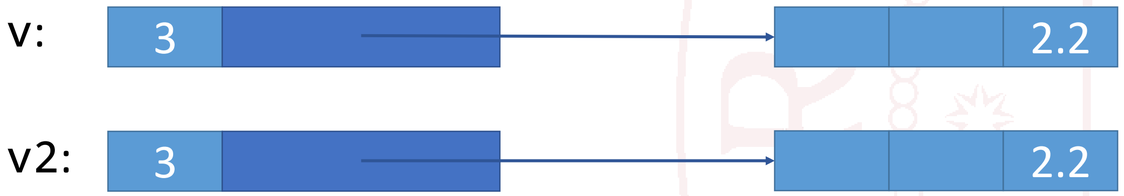
Costruttore di copia

Il costruttore di copia definisce il numero di elementi e alloca la memoria

```
vector::vector(const vector& arg)
    : sz{arg.sz}, elem{new double[arg.sz]}
{
    std::copy(arg.elem, arg.elem+sz, elem);
}
```

- Ora il comportamento è diverso – **deep copy**:

```
vector v2 = v;
vector v2 {v};     // equivalente
```



- Il costruttore di copia viene chiamato quando cerchiamo di inizializzare un `vector` con un altro `vector`

Assegnamento di copia

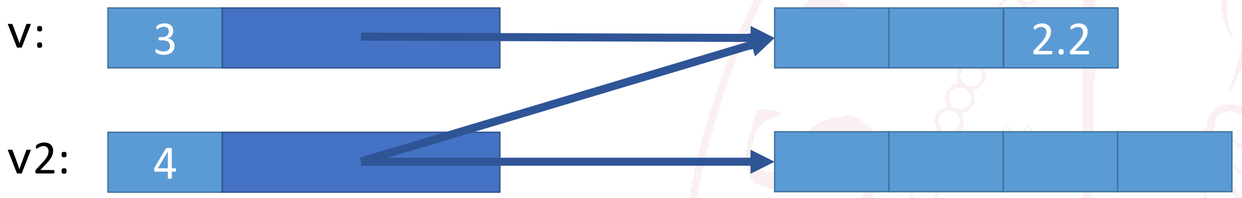
- Un problema analogo al precedente si verifica con l'assegnamento:

```
void f2(int n)
{
    vector v(3);
    v.set(2, 2.2);
    vector v2(4);
    v2 = v;                    // Notare che è un assegnamento!
}
```

- Comportamento di default: copia membro a membro

Problema dell'assegnamento di copia

```
void f2(int n)
{
    vector v(3);
    v.set(2, 2.2);
    vector v2(4);
    v2 = v;
}
```



- Che problema crea?

Stesso problema di prima: i due vettori non sono indipendenti!

- **Doppia chiamata al distruttore**
- **Inoltre, perdita riferimento alla memoria allocata da v2**

Assegnamento di copia

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

Assegnamento di copia

Come potrebbe essere implementata questa funzione membro? (3 min)

1.
2.
3.
4.
5.
6.

Assegnamento di copia

Come potrebbe essere implementata questa funzione membro? (3 min)

```
vector& vector::operator=(const vector& a)
{
    double* p = new double[a.sz];    // 1. alloca nuovo spazio
    copy(a.elem, a.elem+a.sz, p);    // 2. copia gli elementi
    delete[] elem;                   // 3. de-alloco vecchio spazio
    elem = p;                        // 4. resetto puntatore
    sz = a.sz;                       // 5. resetto dimensione
    return *this;                    // 6. ritorno self-reference
}
```

Assegnamento di copia

- L'assegnamento è leggermente diverso dal costruttore
  - Deve gestire i dati vecchi
- Il procedimento proposto:
  - Creare una copia dei dati di source
  - Eliminare gli elementi vecchi
  - Far puntare `elem` ai dati nuovi
- Perché non liberare i dati vecchi *prima*, per risparmiare un puntatore?
  - Non è una buona idea eliminare dei dati finché non siamo sicuri di poterli rimpiazzare

Assegnamento di copia

- Nell'assegnamento di copia dobbiamo tenere conto un caso particolare: l'**auto assegnamento**

```
v = v;
```

- Pur non essendo molto utile, deve essere gestito correttamente dalla classe
- Tipico caso che dimostra che non bisogna eliminare i dati finché non siamo sicuri di poterli rimpiazzare

Copia – terminologia

- **Shallow copy** (copia superficiale)
  - Copia di puntatori o reference, senza copiare i dati
- **Deep copy** (copia profonda)
  - Copia dei dati
  - Sono definiti costruttore di copia e assegnamento di copia
  - Comportamento predefinito di `std::vector`, `std::string`, …
- Prima abbiamo trasformato la shallow copy in deep copy

Copia – terminologia

- Tipi che implementano una shallow copy hanno una *pointer semantics* (o *reference semantics*)
- Tipi che implementano una deep copy hanno una *value semantics*