



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Copia di oggetti

Stefano Ghidoni



# Agenda

- Copia
  - Shallow vs deep
- 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?



# Richiamo – vector

- Riprendiamo il nostro vector

```
class vector {  
    int sz;  
    double* elem;  
  
public:  
    vector(int s)  
        : sz{s}, elem{new double[s]} { /* ... */ }  
    ~vector()  
        { delete[] elem; }  
    // ... - contiene set() per scrivere sul vettore  
};
```



- Cosa succede se copio un vector?

```
void f(int n)
{
}
}
```



- Cosa succede se copio un vector?

```
void f(int n)
{
    vector v(3);

}
```



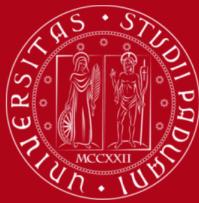


- Cosa succede se copio un vector?

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

}
```





- Cosa succede se copio un vector?

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





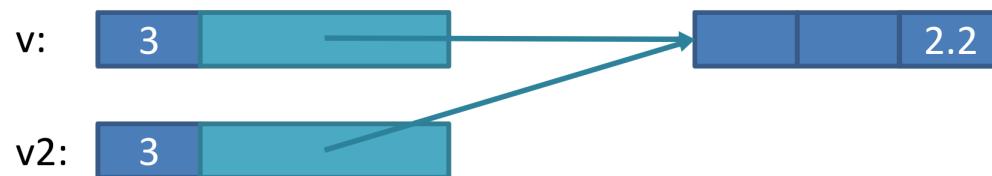
# Copia

- Cosa succede se copio un vector?

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

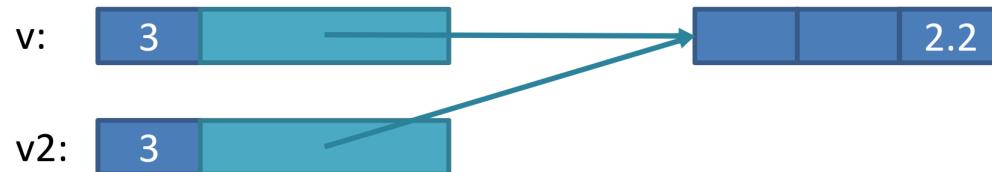
```
// Recall
class vector {
    int sz;
    double* elem;
    // ...
};
```

- Default: copia membro a membro





# Copia



- `v2` non ha una copia degli elementi: *condivide gli stessi elementi* di `v` – **shallow copy**
- **Crea problemi?**  
Si potrebbe pensare di avere due vettori scollegati, invece modificarlo da "una parte" lo modifica anche dall'altra.  
In più modificare `sz` da un lato, non lo modifica dall'altro, quindi si ritroverà disallineato.
- In uscita da `f`, sono chiamati i distruttori di `v` e `v2`
  - È un problema?  
Si, è la doppia deallocazione



# Costruttore di copia

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

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



# Costruttore di copia

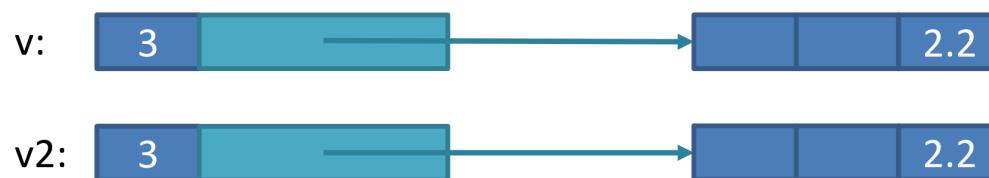
```
vector::vector(const vector& arg)
    : sz{arg.sz}, elem{new double[arg.sz]}
{
    copy(arg.elem, arg.elem+sz, elem);
}
```



Uso l'aritmetica dei puntatori

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

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





# Assegnamento di copia

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

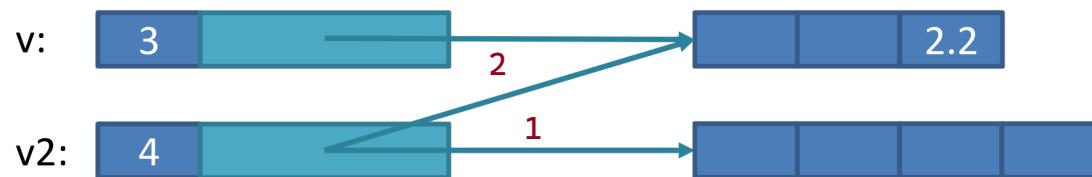
```
void f2(int n)
{
    vector v(3);
    v.set(2, 2.2);
    vector v2(4);
    v2 = v; Il comportamento di default dell'operatore = è di usare il costruttore copia
}
```

- Comportamento di default: copia membro a membro



# Problema della copia

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



- Che problema crea?



# 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)

```
vector& operator=(const vector& a){  
    ???  
    Non posso usare delete e non posso usare new (non posso autoallocarmi dinamicamente)  
}
```

Non è vero, il prof li ha usati nel codice dopo.



# Assegnamento di copia

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

```
vector& vector::operator=(const vector& a)
{
    double* p = new double[a.sz];      // alloca nuovo spazio
    copy(a.elem, a.elem+a.sz, p);      // copia gli elementi
    delete[] elem;
    elem = p;
    sz = a.sz;
    return *this;                      // rit. self-reference
}
```



# Assegnamento di copia

- L'assegnamento è leggermente diverso dal costruttore
  - **Deve gestire i dati vecchi**
- 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 *Una copia onerosa*
  - 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*) è come copiare il puntatore
- Tipi che implementano una deep copy hanno una *value semantics*



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Copia di oggetti

Stefano Ghidoni