

# UNIVERSITÀ DEGLI STUDI DI PADOVA

Elementi essenziali di una buona interfaccia

Stefano Ghidoni





# Agenda

- Regole di progettazione
  - Costruttori
  - Assegnamenti
  - Distruttore

## Operazioni essenziali

- Le operazioni essenziali da considerare per una classe sono:
  - Costruttore con uno o più argomenti
  - Costruttore di default
  - Costruttore di copia (copy constructor)
  - Assegnamento di copia (copy assignment)
  - Costruttore di spostamento (move constructor)
  - Assegnamento di spostamento (move assignment)
  - Distruttore

## Costruttore di default

• Già discusso, ma è importante osservare che

```
vector<double> vi(10);
vector<string> vs(10);
vector<vector<int>> vvi(10);
```

funzionano solo perché esistono i costruttori di default di:

- double
- string
- vector<int> e int

### Distruttore

Necessario se acquisiamo risorse (che devono essere liberate)

### • Risorse:

- Memoria nel free store (allocata dinamicamente)
- File
- Lock, thread handles, socket
- Probabilmente è necessario se ci sono membri puntatori

## Copia e spostamento

- Se è necessario un distruttore, probabilmente sono necessari anche:
  - Costruttore di copia
  - Assegnamento di copia
  - Costruttore di spostamento
  - Assegnamento di spostamento
- Perché ci sono risorse da gestire

## Costruttori e conversioni

- Un costruttore che riceve un singolo argomento definisce una conversione da quell'argomento alla classe
- Può essere utile:

```
class complex {
public:
    complex(double); // conversione da double a complex
    complex(double, double);
    // ...
};

complex z1 = 3.14; // ok: conversione
complex z2 = complex{1.2, 3.4};
```

### Costruttori e conversioni

- A volte la conversione è indesiderata
  - Es: la nostra classe vector ha un costruttore che accetta un int, usato per costruire vettori di n elementi

# Costruttori espliciti

- Conversione indesiderata?
  - È possibile eliminare le conversioni implicite

```
class vector {
   // ...
   explicit vector(int);
   // ...
vector v = 10;
                  // errore
v = 20;
                    // errore
vector v0(10);
                    // ok!
void f(const vector&);
f(10);
                     // errore
f(vector(10));
                     // ok!
```

## Analizzare costruttori e distruttori

- Un costruttore può essere chiamato in modi e momenti diversi
- Costruttore chiamato quando si crea un oggetto
  - Oggetto inizializzato
  - New
  - Oggetto copiato
- Distruttore chiamato quando
  - Un nome esce dallo scope
  - È usato delete

## Analizzare costruttori e distruttori

- Esploriamo usando questa struct
- Quali elementi visti poco fa riconoscete?

```
struct X {
    int val;
    void out(const string& s, int nv)
        { cerr << this << "->" << s << ": " << val << "("
          << nv << ")\n"; }
   X() \{ out("X()", 0); val = 0; \}
    X(int v) { val = v; out("X(int)", v); }
    X(const X\& x) \{val = x.val; out("X(X\&)", x.val); \}
    X& operator=(const X& a) { out("X::operator=()", a.val);
                                val = a.val; return *this; }
    ~X() { out("~X(), 0); }
```

## Analizzare costruttori e distruttori

- Esploriamo usando questa struct
- Quali elementi visti poco fa riconoscete?

```
struct X {
    int val;
    void out(const string& s, int nv)
         { cerr << this << "->" << s << ": " << val << "("
           << nv << ")\n"; }
    X() \{ out("X()", 0); val = 0; \}
                                                                Costruttore di default
    X(int v) { val = v; out("X(int)", v); }
    X(const X\& x) \{val = x.val; out("X(X\&)", x.val); \}
                                                                Costruttore di copia
    X& operator=(const X& a) { out("X::operator=()", a.val)
                                   val = a.val; return *this; }
                                                                    to di copia
    ~X() { out("~X(), 0); }
                                                                  Distruttore
```

#### Esercizio

Per casa: implementare ed eseguire:

```
X \text{ glob}(2);
                               // variabile globale
X copy(X a) { return a; }
X copy2(X a) { X aa = a; return aa; }
X& ref_to(X& a) { return a; }
X* make(int i) { X a(i); return new X(a); }
struct XX { X a; X b; };
// segue
```

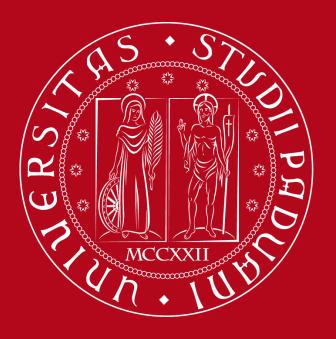
### Esercizio

```
int main()
{
                              // var locale
    X loc \{4\};
    X loc2 {loc};
                             // costruttore di copia
                              // assegnamento di copia
    loc = X{5};
    loc2 = copy(loc);
                             // call by value e return
    loc2 = copy2(loc);
    X loc3 {6};
    X& r = ref_to(loc);
                             // call by reference e return
    delete make(7);
    delete make(8);
    vector<X> v(4);
    XX loc4;
    X* p = new X{9};
    delete p;
    X* pp = new X[5];
    delete[] pp;
```



## Note all'esercizio

- In funzione del compilatore usato, alcune copie di copy e copy2 potrebbero non essere eseguite
- Una copia di un oggetto non utilizzato può non essere eseguita
  - Il compilatore è autorizzato a ritenere che un costruttore di copia esegua solamente una copia, e nient'altro



# UNIVERSITÀ DEGLI STUDI DI PADOVA

Elementi essenziali di una buona interfaccia

Stefano Ghidoni

