



# UNIVERSITÀ DEGLI STUDI DI PADOVA

## Smart pointer

Stefano Ghidoni



- Smart pointer: concetto
- I due tipi principali di smart pointer
- Esempi



- Puntatori intelligenti
- Classi template definite in `<memory>`
- Gestiscono automaticamente la deallocazione della memoria quando escono dallo scope
  - Eliminano memory leak
  - Eliminano dangling pointer
- Un overhead è presente, ma **molto limitato** rispetto a un garbage collector



- Esistono vari tipi di smart pointer
  - `unique_ptr`
  - `shared_ptr`
  - ~~`auto_ptr` (deprecated)~~
- Diversi per alcune caratteristiche fondamentali che ne regolano l'uso
- Si è soliti dire che uno smart pointer *detiene* un puntatore



- unique\_ptr è uno smart pointer che:
  - Dealloca la memoria uscendo dallo scope
  - Permette i move
  - **Non permette le copie**
    - Come si ottiene questo effetto?



- unique\_ptr è uno smart pointer che:
  - Dealloca la memoria uscendo dallo scope
  - Permette i move
  - **Non permette le copie**
    - Costruttore e assegnamento di copia disabilitati
  - **Non ha sostanziali overhead** rispetto a un puntatore



- Inizializzazione
  - Nel costruttore, oppure:
  - Funzione reset
- Re-inizializzazione: funzione reset
  - Fa sì che unique\_ptr detenga un nuovo puntatore
  - Se unique\_ptr deteneva un puntatore prima della chiamata, l'oggetto puntato è distrutto
  - Senza argomenti: semplice deallocazione



- Accesso
  - È possibile usare gli operatori \* e -> come se fosse un puntatore
- Liberazione automatica della memoria
  - Quando unique\_ptr è distrutto, cancella l'oggetto puntato!
- Rilascio del puntatore
  - release() *estrae* il puntatore dallo unique\_ptr, il quale è resettato a nullptr





- Esempio di utilizzo – riscriviamo make\_vec:

```
vector<int>* make_vec()
{
    unique_ptr<vector<int>> p { new vector<int> };
    // ... riempimento del vettore con i dati - può
    // lanciare un'eccezione!
    return p.release();
}
```

- Inizializzazione nel costruttore
- Release del puntatore:
  - A fine utilizzo
  - Quando è lanciata un'eccezione
    - Quindi in make\_vec **non è necessario fare il catch!**



- Esempio di utilizzo – riscriviamo make\_vec:

```
vector<int>* make_vec()
{
    unique_ptr<vector<int>> p { new vector<int> };
    // ... riempimento del vettore con i dati - può
    // lanciare un'eccezione!
    return p.release();
}
```

- `release()` *estrae* il puntatore dallo `unique_ptr`, il quale è resettato a `nullptr`
  - Quindi la successiva distruzione di `p` non cancella nulla
  - Si perde la deallocazione automatica



- Nota: allocare dinamicamente uno `std::vector` è solitamente **dannoso**
  - Usato come esempio perché il riempimento può causare un'eccezione

```
vector<int>* make_vec()
{
    unique_ptr<vector<int>> p { new vector<int> };
    // ... riempimento del vettore con i dati - può
    // ritornare un'eccezione!
    return p.release();
}
```



- Uno `unique_ptr`:
  - Non può essere copiato
  - Può essere spostato!
- È possibile ritornare uno `unique_ptr`
  - Sarà usato il move constructor

```
unique_ptr<int> AllocateAndReturn()  
{  
    unique_ptr<int> ptr_to_return(new int(42));  
    return ptr_to_return;  
}
```



- La copia è invece disabilitata

```
int main()
{
    unique_ptr<int> moved_ptr = AllocateAndReturn();
    // Errore di compilazione!
    // unique_ptr<int> copied_ptr = moved_ptr;

    cout << *moved_ptr << endl;

    return 0;
}
```



- Possiamo usare `unique_ptr` anche per risolvere il problema della delete nel chiamante
  - Basta ritornare lo `unique_ptr`!

```
unique_ptr<vector<int>> make_vec()
{
    unique_ptr<vector<int>> p { new vector<int> };
    // ... riempimento del vettore con i dati - può
    // ritornare un'eccezione!
    return p;
}
```



- `unique_ptr` ha una restrizione: due `unique_ptr` non possono puntare allo stesso oggetto
  - Come detto, la copia tra `unique_ptr` è disabilitata

```
void no_good()
{
    unique_ptr<X> p { new X };
    unique_ptr<X> q { p };    // errore
} // qui sia p che q eliminerebbero l'oggetto
```



- Inizializzare due `unique_ptr` con lo stesso puntatore è un errore
  - Doppia delete
  - Non rilevabile dal compilatore

```
void no_good()
{
    int* p = new int;
    unique_ptr<int> sp {p};
    unique_ptr<int> sp2 {p}; // errore logico, ma compila
} // qui sia sp che sp2 eliminano l'oggetto
```





- Inizializzare due `unique_ptr` con lo stesso puntatore è un errore
  - Doppia delete
  - Non rilevabile dal compilatore
- L'esistenza del puntatore da incapsulare è un punto debole
  - Risolvibile con `make_unique`



# Shared pointer

- Lo shared pointer rimuove i vincoli di `unique_ptr`
- Uno `shared_ptr`:
  - Può essere copiato
  - Può essere condiviso – molti `shared_ptr` possono detenere lo stesso puntatore



- Questo è possibile tramite il reference counting
  - shared\_ptr che detengono lo stesso puntatore "si conoscono"
- La memoria è deallocata quando **l'ultimo** shared pointer che detiene la memoria è distrutto
- Questo meccanismo **consuma risorse**



- AllocateAndReturn: versione con shared\_ptr

```
shared_ptr<int> AllocateAndReturn()
{
    shared_ptr<int> ptr_to_return(new int(42));

    return ptr_to_return;
}
```



- Con `shared_ptr` la copia è possibile

```
int main()
{
    shared_ptr<int> moved_ptr = AllocateAndReturn();

    // Questo è possibile!
    shared_ptr<int> copied_ptr = moved_ptr;

    cout << *moved_ptr << endl;

    return 0;
}
```



- Concetto di smart pointer
- `unique_ptr`
- `shared_ptr`
- Esempi – riscrivere `make_vec` con `unique_ptr`



# UNIVERSITÀ DEGLI STUDI DI PADOVA

## Smart pointer

Stefano Ghidoni