I container della STL: all'interno di `std::vector`

Valentino Picotti

27 maggio 2015

1. Introduzione

2. Standard Template Library
   - Template
   - Container e algoritmi standard
   - Iteratori

3. Case study: `std::vector`

Language Overview

General-purpose programming language:

- Imperativo
- Programmazione Orientata agli Oggetti
- Programmazione Generica

Main Concepts

- Value Semantics
- Concrete Types
- Operator Overloading

Value semantics

C++:

```
int x = 42;
int y = x;

vector<int> x(3);
vector<int> y = x;
```

Java:

```
int x = 42;
int y = x;

ArrayList<int> x = new ArrayList<int>(3);
ArrayList<int> y = x;
```

Qual è la differenza?

Value Semantics - Precisazione

```
std::vector<int> myFunction(...)
{
  std::vector<int> v = ...;

  return v;
}

vector<int> y = myFunction(...);
```

Quante copie sto facendo?

Concrete Types

Tipi definiti dall'utente, distinti da classi astratte e loro gerarchie.
Vantaggi:

- Nessuna indirezione attraverso i puntatori
- Nessun overhead in termini di spazio
- Piena integrazione nel linguaggio

Operator Overloading

Il comportamento di un operatore dipende dal tipo dei suoi argomenti.
Vantaggi:

- Supporto sintattico pari ai tipi built-in
- Notazione concisa per le operazioni più comuni
- Notazione convenzionale al tipo di dato (e.g. dati algebrici)

Operator Overloading

```
std::vector<int> x = ...;
int y = x[0];

glm::vec4 v1 = ...;
glm::vec4 v2 = ...;
glm::mat4 m = ...;

v2 = m * v1;
```

La Standard Template Library

La libreria standard del C++ include, tra le altre cose, la Standard Template Library.

- Collezione di strutture dati implementate come generiche classi template, chiamati *container*
- Algoritmi generici che operano sui container

I template

Meccanismo con cui il C++ supporta il paradigma della *programmazione generica*.

```
template<typename T>
T max(T v1, T v2) {
  return v1 > v2 ? v1 : v2;
}
```

Vari elementi del linguaggio si possono parametrizzare:

- Funzioni template
- Classi template
- Variabili template, ecc...

La Standard Template Library

I container standard sono implementazioni generiche di strutture dati di uso comune:

- Sequence containers:
  `std::array`, `std::vector`, `std::deque`, `std::list`, ecc...
- Associative containers (ordered and unordered):
  `std::map`, `std::set`, `std::unordered_map`, `std::unordered_set`, ecc...
- Adapters:
  `std::stack`, `std::queue`, `std::priority_queue`

La Standard Template Library

Algoritmi standard:

- Funzioni template che implementano algoritmi di base: es. `std::sort`, `std::transform`, `std::accumulate`, `std::equal`, `std::all_of`
- Generici e riutilizzabili

Iteratori

Gli iteratori sono l'interfaccia comune con cui manipolare genericamente tutti i container della STL.

- Ogni container fornisce una coppia di iteratori `begin()`/`end()`
- Gli iteratori astraggono il container presentandolo come una generica sequenza di elementi.
- Tramite un iteratore si può accedere agli elementi della sequenza ...
- ... e muoversi nella sequenza stessa.
- Gli algoritmi standard manipolano sequenze di elementi identificate da coppie di iteratori.

Iteratori

```
std::vector<int> v = { .... };

int max = v[0];

for(int i = 1; i < v.size(); ++i) {
  if(v[i] > max)
    max = v[i];
}

std::cout << max;
```

Iteratori

```
std::vector<int> v = { .... };

std::cout << *std::max_element(v.begin(), v.end());
```

Iteratori

```
template<typename Iterator>
Iterator max_element(Iterator begin, Iterator end)
{
  Iterator max = begin;

  for(Iterator it = v.begin(); it != v.end(); ++it) {
    if(*it > *max)
      max = it;
  }

  return max;
}
```

Case study: `std::vector`

Container che incapsula un array di dimensione dinamica

- Elementi contigui in memoria
- Gestione automatica della memoria

Operazioni comuni:

- Accesso diretto - costante $\mathcal{O}(1)$
- Inserimenti e rimozioni alla fine - ammortizzato $\mathcal{O}(1)$
- Inserimenti e rimozioni - lineare nella distanza dalla fine $\mathcal{O}(n)$

Ridimensionamento dinamico

Implementazione:

- Puntatore a un array allocato dinamicamente
- Membro *size* : numero di elementi contenuti nel vector
- Membro *capacity* : numero massimo di elementi che l'array sottostante può contenere

Ridimensionamento dinamico

Una certa quantità di memoria viene preallocata, in modo che non serva allocarne di nuova ad ogni ridimensionamento.

- `resize(n)` ridimensiona il vector
- `reserve(n)` gestisce la memoria preallocata

Esempio

```
{
  std::vector<int> v = {1, 2, 3, 4};

  v[0] = 42;  // Sintassi array-like

  // Dimensione dinamica
  v.push_back(5);
  v.resize(20);

} // memoria rilasciata automaticamente
```

Ridimensionamento dinamico

Modificare la dimensione del vector comporta tre passi:

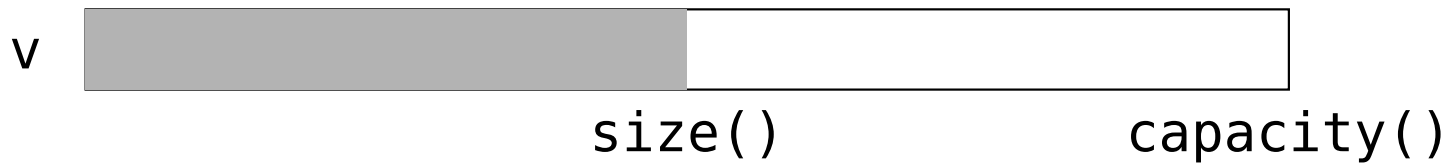
- Allocare un nuovo array della dimensione giusta
- Copiare tutto il contenuto nel nuovo array
- Deallocare la memoria vecchia

Va ovviamente evitato di farlo ad ogni `push_back()`.

Ridimensionamento dinamico

```
std::vector<int> v;

v.reserve(20);
v.resize(10);
```



Ridimensionamento dinamico

Quando il vector viene ridimensionato oltre l'attuale `capacity()`:

- Viene allocato un nuovo array di dimensione $k \cdot$ `capacity()`
- k costante, solitamente 1.5 o 2.
- Ciò garantisce a `push_back()` una complessità di $\mathcal{O}(1)$ ammortizzato

Complessità di push_back()

Se l'n-esimo push_back() causa una riallocazione abbiamo già dovuto riallocare memoria $\log_k(n)$ volte.

- La i-esima riallocazione è costata $\mathcal{O}(k^i)$, quindi in totale:

$$\sum_{i=1}^{\log_k(n)} k^i = \frac{k - k^{\log_k(n)+1}}{1-k} \quad \text{serie geometrica}$$

$$= \frac{k(1 - k^{\log_k(n)})}{1-k} = \frac{k(1-n)}{1-k}$$

- Distribuito su n operazioni:

$$\frac{k(1-n)}{n(1-k)} \cong \frac{k}{1-k} = \mathcal{O}(1)$$

Esempio

Un sguardo al codice