Linguaggi di Programmazione

Ivano Salvo

OOP6: More on C++

Corso di Laurea in **Matematica**

Lezione **9**, 24 novembre 2020

Appendice alla Lezione 7

Ancora sulle STL

Imparare a usare le risorse WEB

cplusplus.com/reference/vector/vector/

class template

std::vector

<vector>

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

Vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

Specifica dei Metodi divisi in gruppi

fx Member functions

(constructor)	Construct vector (public member function)
(destructor)	Vector destructor (public member function)
operator=	Assign content (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)

Capacity:

size	Return size (public member function)
max_size	Return maximum size (public member function)
resize	Change size (public member function)
capacity	Return size of allocated storage capacity (public member function)
empty	Test whether vector is empty (public member function)
reserve	Request a change in capacity (public member function)
shrink_to_fit C++11	Shrink to fit (public member function)

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

Specifica dei Metodi divisi in gruppi

Element access:

operator[]	Access element (public member function)
at	Access element (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
data C++11	Access data (public member function)

accesso diretto via indice

accesso al primo/ultimo

Modifiers:

assign	Assign vector content (public member function)
push_back	Add element at the end (public member function)
pop_back	Delete last element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

riempimento di molti valori

inserimento/eliminazione in coda

Allocator:

get_allocator	Get allocator (public member function)

fx Non-member function overloads

relational operators	Relational operators for vector (function template)
swap	Exchange contents of vectors (function template)

Template specializations

vector<bool>	Vector of bool (class template specialization)

Specifica dei Metodi divisi in gruppi

std::vector::assign

C++98 | C++11

possibili usi (overloading)

range (1)	`template <class InputIterator>` `void assign (InputIterator first, InputIterator last);`
fill (2)	`void assign (size_type n, const value_type& val);`

esempi d'uso

Example

```
// vector assign
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> first;
  std::vector<int> second;
  std::vector<int> third;

  first.assign (7,100);             // 7 ints with a value of 100

  std::vector<int>::iterator it;
  it=first.begin()+1;

  second.assign (it,first.end()-1); // the 5 central values of first

  int myints[] = {1776,7,4};
  third.assign (myints,myints+3);   // assigning from array.

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  std::cout << "Size of third: " << int (third.size()) << '\n';
  return 0;
}
```

std:: mettendo a inizio programma using namespace std

assign può copiare un pezzo di un vector dentro un altro, usando gli iteratori per scorrere… (Algoritmi)

Specifica dei Metodi divisi in gruppi

Return Value

none

If a reallocation happens, the storage is allocated using the container's allocator, which may throw exceptions on failure (for the default allocator, `bad_alloc` is thrown if the allocation request does not succeed).

possibili eccezioni: comportamenti anomali

Example

```
// resizing vector
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;

  // set some initial content:
  for (int i=1;i<10;i++) myvector.push_back(i);

  myvector.resize(5);
  myvector.resize(8,100);
  myvector.resize(12);

  std::cout << "myvector contains:";
  for (int i=0;i<myvector.size();i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

Edit & Run

vedete che si vedono diversi casi: resize che diminuisce, resize che aumenta (riempendo con un valore) o aumenta con valore di default.

Output:

```
myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

Uno sguardo alle deque

Modifiers:

assign	Assign container content (public member function)
push_back	Add element at the end (public member function)
push_front	Insert element at beginning (public member function)
pop_back	Delete last element (public member function)
pop_front	Delete first element (public member function)
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_front C++11	Construct and insert element at beginning (public member function)
emplace_back C++11	Construct and insert element at the end (public member function)

Si possono aggiungere/togliere elementi
sia in testa che in coda
(push_back/pop_back e push_front/pop_front

Inserzioni di elementi “in mezzo”

public member function

std::deque::emplace

<deque>

```
template <class... Args>
  iterator emplace (const_iterator position, Args&&... args);
```

Construct and insert element

The container is extended by inserting a new element at *position*. This new element is constructed in place using *args* as the arguments for its construction.

This effectively increases the container size by one.

Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in list or forward_list containers. See emplace_front and emplace_back for member functions that extend the container directly at the beginning or at the end.

The element is constructed in-place by calling allocator_traits::construct with *args* forwarded.

Parameters

`position`
: Position in the container where the new element is inserted.
Member type `const_iterator` is a random access iterator type that points to a constant element.

`args`
: Arguments forwarded to construct the new element.

Return value

An iterator that points to the newly emplaced element.

Deque: costruttori

C++98 | C++11

(1) empty container constructor (default constructor)
Constructs an empty container, with no elements.

(2) fill constructor
Constructs a container with *n* elements. Each element is a copy if *val*.

(3) range constructor
Constructs a container with as many elements as the range `[first,last)`, with each element constructed from its corresponding element in that range, in the same order.

(4) copy constructor
Constructs a container with a copy of each of the elements in *x*, in the same order.

The container keeps an internal copy of *alloc*, which is used to allocate storage throughout its lifetime. If no *alloc* argument is passed to the constructor, a default-constructed allocator is used, except in the following case:
- The copy constructor *(4)* creates a container that keeps and uses a copy of *x*'s allocator.

The storage for the elements is allocated using this internal allocator.

Discussione: Stack sottoclasse Vector?

La classe `Vector` ha la possibilità di aggiungere in fondo al vettore. Sarebbe corretto derivare la classe `Stack` da `Vector` ereditando i metodi `push_back`? e `pop_back`?

Se SI perché, se NO qual è il modo giusto di procedere?

NO. Anche se questa (più o meno è la soluzione nelle Collection Classes di SmallTalk e di Java). Sarebbe in tal caso **necessario oscurare** gli altri possibili accessi/modificatori al `Vector` per garantire l'interfaccia di `Stack`.

La soluzione corretta è fare una classe `Stack` che **ha dentro un oggetto di tipo `Vector`** e implementare il push/pop dello `Stack` usando i metodi di `Vector` (**wrapping**).

Lezione 9a

Reference & Puntatori

C++ vs C: the clean & the dirty

Tra gli obbiettivi di C++ c'è "**ripulire**" il C di alcuni dettagli "scabrosi".

Ad esempio, **il tipo `bool` del C++ vuole evitare il malcostume C di usare un qualsiasi tipo di dato nelle condizioni logiche** dove 0 (interi), `NULL` (puntatori), carattere 0 (codice asci 0) valgono False, mentre tutti gli altri valori valgono True.

Il **C++ proibisce** anche la promiscuità indotta dal **tipo `void*`**, che essendo compatibile per assegnazione con ogni altro tipo pointer, permette di fatto di **bypassare ogni controllo di tipo**.

Aspetto **critico della compatibilità tra C++ e C**: per compilare in C++ assegnazioni via void* occorre aggiungere un **cast esplicito**.

In questa direzione, in C++ esiste una forma edulcorata ("fancy" secondo Bruce Eckel) di pointer, chiamata **reference**.

Riflessione: il tipo `void*` vi permetterebbe di definire strutture dati generiche, ma quali sono le differenze con i templates o il polimorfismo di Haskell?

References

Le reference di C++ sono mutuate dai puntatori come vengono trattati in Algol, Pascal e (in un certo senso) Java.

- Le reference **puntano a memoria allocata**: non possono mai produrre un errore di Segmentation Fault.
- **Devono essere inizializzate**: tipicamente al pointer di una variabile già definita.
- Vengono **dereferenziate automaticamente**, senza bisogno di usare l'operatore `*`.

```
int main(){
   int a=3;
   int & ref=a; /* punta alla memoria di a */
   ref++; /* incrementa il valore di a */
   printf("%d\n",a); /* stampa 4 */
   printf("%d\n",ref); /* stampa 4 */
}
```

References: parametri delle funzioni

Bruce Eckel definisce il passaggio di parametri di C++ "**clean but hidden**" e quello di C "**ugly but explicit**". Ma è vero?

vediamo l'esempio della scambia:

```
void scambiaCpp(int& x, int& y){
    int h=x;
    x=y;
    y=h;
}
/* chiamata */
scambiaCpp(a, b);
```

Il chiamante non ha nessuna informazione sul fatto che `scambiaCpp` *può produrre side-effects su a e b*

Dentro `scambiaC` *sono costretto a dereferenziare i pointer* `x` *e* `y`*. La chiamata esplicita i possibili side-effects*

```
void scambiaC(int* x, int* y){
    int h=*x;
    *x=*y;
    *y=h;
}
/* chiamata */
scambiaC(&a, &b);
```

Constant References

Una curiosa (ma utilissima possibilità) è quella di passare parametri per **referenza costante**.

È utile in C++ in quanto se passo grandi oggetti, la semantica call-by-value costringe a copiarli. Così otteniamo l'effetto della call-by-value evitando di ricopiare dati.

```
void g(const int& x){
   x++; /* proibito, x è const */
}
void h(int &x){x++;}
void f(const int& x){
   printf("%d\n",x);
}
/* chiamate */
h(3); /* proibita: h necessita di un
l-value, 3 è un valore temporaneo */

f(3); /* corretto */
```

Lezione 9b

Copy Constructor e Distruttori

Copia di Oggetti

C++ permette di allocare oggetti sullo stack e passare oggetti per valore. Questo è contrario **all'OOP classico** (**Java** e **SmallTalk**), dove **gli oggetti sono sempre riferiti via puntatori** e quindi, anche **sempre passati per riferimento**.

Tuttavia, per garantire la semantica della call-by-value e dell'assegnazione, è necessario poter copiare oggetti.

Se il programmatore non fa nulla, esiste, associato a ogni classe un **copy-costructor di default**, che si limita a ricopiare la memoria occupata per l'oggetto.

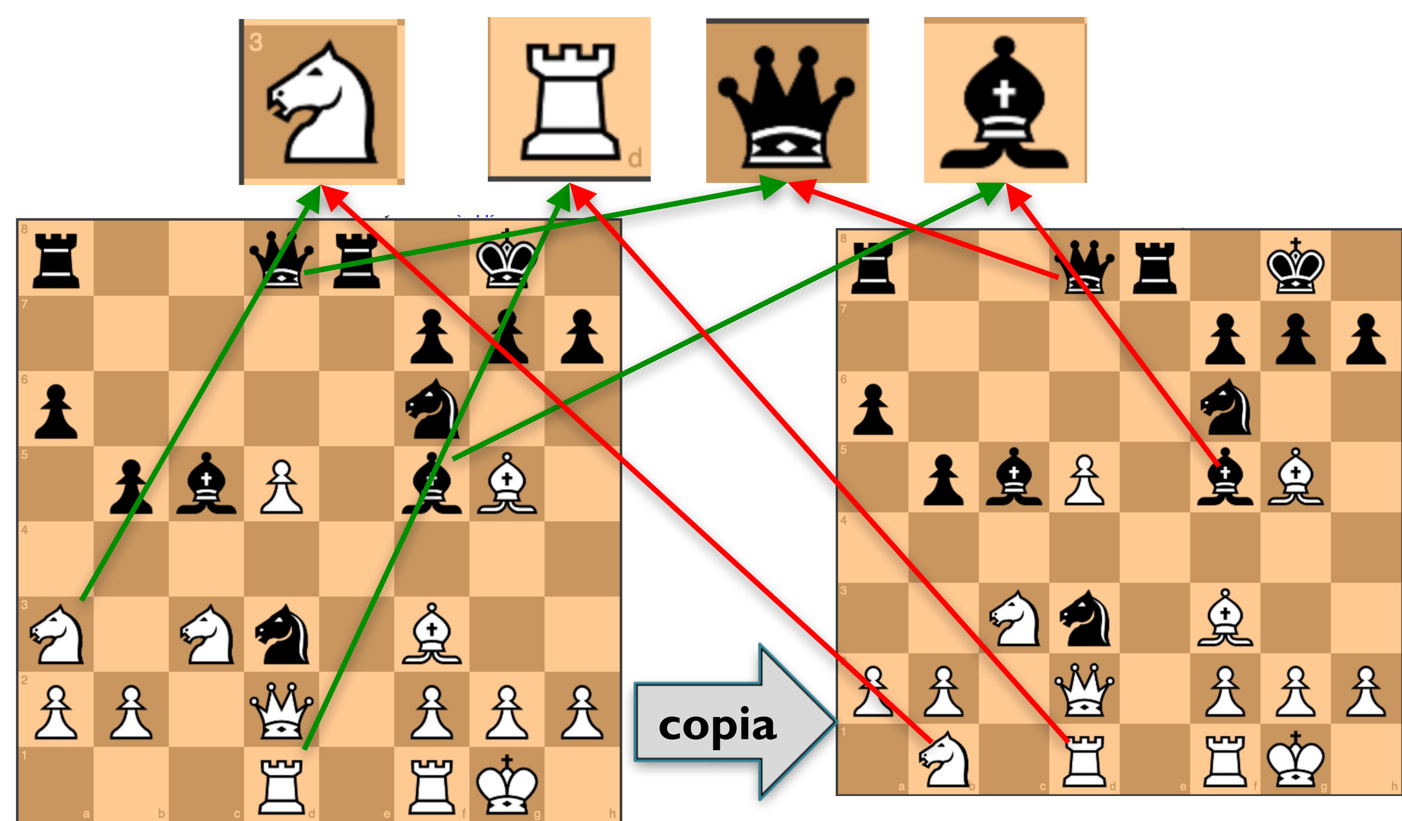
Ma quali sono i problemi?

Copia di Oggetti: Problema 1

Un oggetto potrebbe contenere riferimenti ad altri oggetti.
Copiando semplicemente **la memoria**, **non si garantisce la semantica call-by-value**.

Esempio: la **scacchiera** è una **matrice di puntatori ai pezzi**.

Copiando **creo sharing sugli stessi pezzi**! (che **potrebbe essere ok**)



Copia di Oggetti: problema 2

A volte ci sono **invarianti di tipo di dato** che necessitano un riaggiornamento alla creazione di ogni nuovo oggetto.

Ad esempio, immaginate ci sia una variabile che **conta quanti oggetti sono stati creati**. Se semplicemente copio la memoria, non riaggiorno questa variabile.

Una **variabile statica** appartiene alla classe: c'è un'**unica copia** per tutta la **classe** (serve ad esempio per informazione condivisa).

Un **metodo statico non dipende** dallo stato dell'oggetto, cioè **da variabili di istanza non statiche**)

```
class HowMany{
   static int objectCount;
public:
   HowMany(){objectCount++;}
   static void print(){
      printf("%d\n"),objectCount;
   }
}/* end class */
```

Copy Constructor (1)

Per programmare opportunamente cosa accade alla copia di un oggetto **è necessario definire** un codice più significativo per **il copy-costructor** che ha sempre prototipo: `C(C& c)` dove `C` è il nome della classe.

Il copy-constructor viene chiamato ogni volta che è necessario fare la copia di un oggetto.

Se volete **inibire la copia degli oggetti**, si può **definire il copy-constructor privato**: il copy-constructor definito dal programmatore elimina quello di default, ma è privato!

```
public:
      /* copy constructor */
   HowMany(const Howmany& h){objectCount++;}
   ...
```

Copy Constructor (2)

Nel caso in cui vogliate ricopiare oggetti puntati dentro un oggetto (se fosse necessario) **occorre scrivere codice per clonare tutti gli oggetti di cui si hanno i riferimenti**.

Ovviamente, al solito, ciò che è **giusto** fare **dipende dalla logica** dell'architettura dell'applicazione.

Sperimentazioni: fate produrre output a costruttori e copy-constructor e cercate di capire quando vengono invocati.

Cercate di vedere cosa succede agli oggetti passati come parametro o ritornati come valore (oggetti sullo stack) e agli oggetti creati attraverso una `new`.

Deallocazione di Oggetti

Gli stessi problemi sorgono quando e come deallocare gli oggetti.

Ancora una volta, ci sono deallocazioni automatiche dovute a oggetti allocati sullo stack: quando usciamo da un contesto (ad esempio, una funzione finisce di eseguire) va deallocata tutta la memoria che appartiene a quel contesto.

Ma nel caso della nostra classe HowMany? come garantire la correttezza del conteggio quando vengono deallocati gli oggetti?

Inoltre: la deallocazione di una struttura dati o di un oggetto aggregato deve causare la deallocazione degli oggetti riferiti? Risposta difficile, ma come farla?

Ovviamente, anche in questo caso, il C++ genera per ogni classe un **distruttore di default**.

Distruttore

Per programmare opportunamente cosa accade alla deallocazione di un oggetto **è necessario definire** un codice più significativo per **il distruttore** che ha sempre prototipo:
`~C()` dove `C` è il nome della classe.

Il **distruttore** viene chiamato **automaticamente** sugli oggetti allocati sullo **stack**, **ogni volta che si esce dal contesto** a cui appartengono (tipicamente una funzione o un blocco -- **scope**).

Deve invece essere chiamato esplicitamente per gli oggetti allocati sull'heap per effetto di una new. Si fa usando il comando `delete ref;` dove ref è un puntatore all'oggetto da eliminare.

```
public:
        /* distruttore */
    ~HowMany(){objectCount--;}
    ...
```

Quando e cosa deallocare?

La **deallocazione** di memoria è una delle più **fertili sorgenti di errori** nella programmazione.

Anche se l'uso di `new` e `delete` **migliora la situazione rispetto alle primitive C** `malloc/calloc` e `free` ci sono questioni spesso difficili da dirimere mentre si programma.

Ad esempio: se dealloco la struttura dati che mi è servita a contenere i pezzi per valutare lo scacco al re, devo deallocare anche i pezzi? Qui la risposta è ovviamente No, ma le cose potrebbero non essere sempre chiare.

È viceversa chiaro che se deallocassi la scacchiera, dovrei voler deallocare i pezzi riferiti dalla scacchiera e allocati all'inizio della partita.

Occorre sempre chiedersi (in fase di progettazione) **a chi appartengono gli oggetti** e di chi è la **responsabilità di crearli e distruggerli**.

Lezione 6

That's all Folks…

Grazie per l'attenzione…

…Domande?