C++ Standard Library

From Wikipedia, the free encyclopedia

In the C++ programming language, the **C++ Standard Library** is a collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself.[1] The C++ Standard Library provides several generic containers, functions to utilize and manipulate these containers, function objects, generic strings and streams (including interactive and file I/O), support for some language features, and everyday functions for tasks such as finding the square root of a number. The C++ Standard Library also incorporates 18 headers of the ISO C90 C standard library ending with ".h", but their use is deprecated.[2] No other headers in the C++ Standard Library end in ".h". Features of the C++ Standard Library are declared within the `std` namespace.

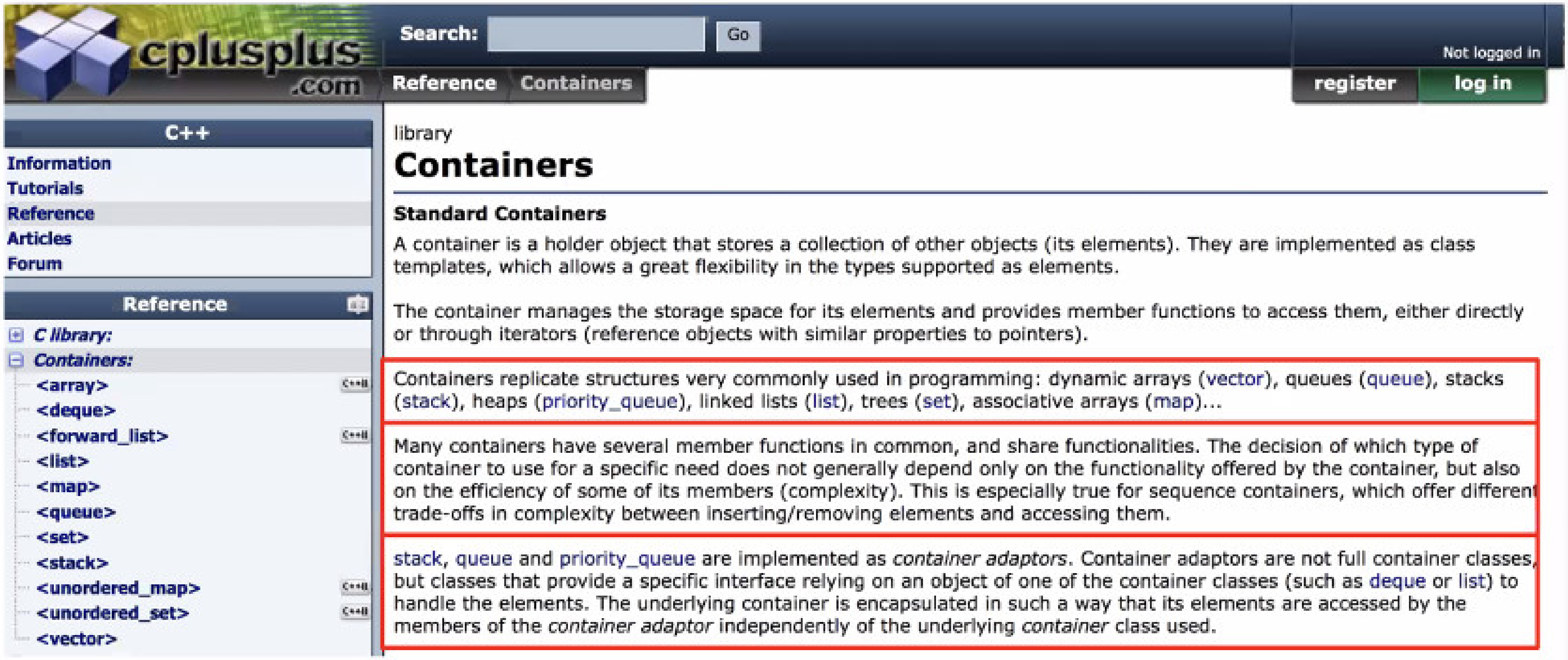
The C++ Standard Library is based upon conventions introduced by the Standard Template Library (STL), and has been influenced by research in generic programming and developers of the STL such as Alexander Stepanov and Meng Lee.[3][4] Although the C++ Standard Library and the STL share many features, neither is a strict superset of the other.

A noteworthy feature of the C++ Standard Library is that it not only specifies the syntax and semantics of generic algorithms, but also places requirements on their performance.[5] These performance requirements often correspond to a well-known algorithm, which is expected but not required to be used. In most cases this requires linear time $O(n)$ or linearithmic time $O(n \log n)$, but in some cases higher bounds are allowed, such as quasilinear time $O(n \log^2 n)$ for stable sort (to allow in-place merge sort). Previously sorting was only required to take $O(n \log n)$ on average, allowing the use of quicksort, which is fast in practice but has poor worst-case performance, but introsort was introduced to allow both fast average performance and optimal worst-case complexity, and as of C++11, sorting is guaranteed to be at worst linearithmic. In other cases requirements remain laxer, such as selection, which is only required to be linear on average (as in quicksort),[6] not requiring worst-case linear as in introselect.

The C++ Standard Library underwent ISO standardization as part of the C++ ISO Standardization effort, and is undergoing further work[7] regarding standardization of expanded functionality.

Lo standard C++ prevede la definizione di:

- Template di classi collezione: **contenitori**
- Template di funzione: **algoritmi generici**



library

Containers

Standard Containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers replicate structures very commonly used in programming: dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map)...

Many containers have several member functions in common, and share functionalities. The decision of which type of container to use for a specific need does not generally depend only on the functionality offered by the container, but also on the efficiency of some of its members (complexity). This is especially true for sequence containers, which offer different trade-offs in complexity between inserting/removing elements and accessing them.

stack, queue and priority_queue are implemented as *container adaptors*. Container adaptors are not full container classes, but classes that provide a specific interface relying on an object of one of the container classes (such as deque or list) to handle the elements. The underlying container is encapsulated in such a way that its elements are accessed by the members of the *container adaptor* independently of the underlying *container* class used.

La documentazione di una libreria è fondamentale

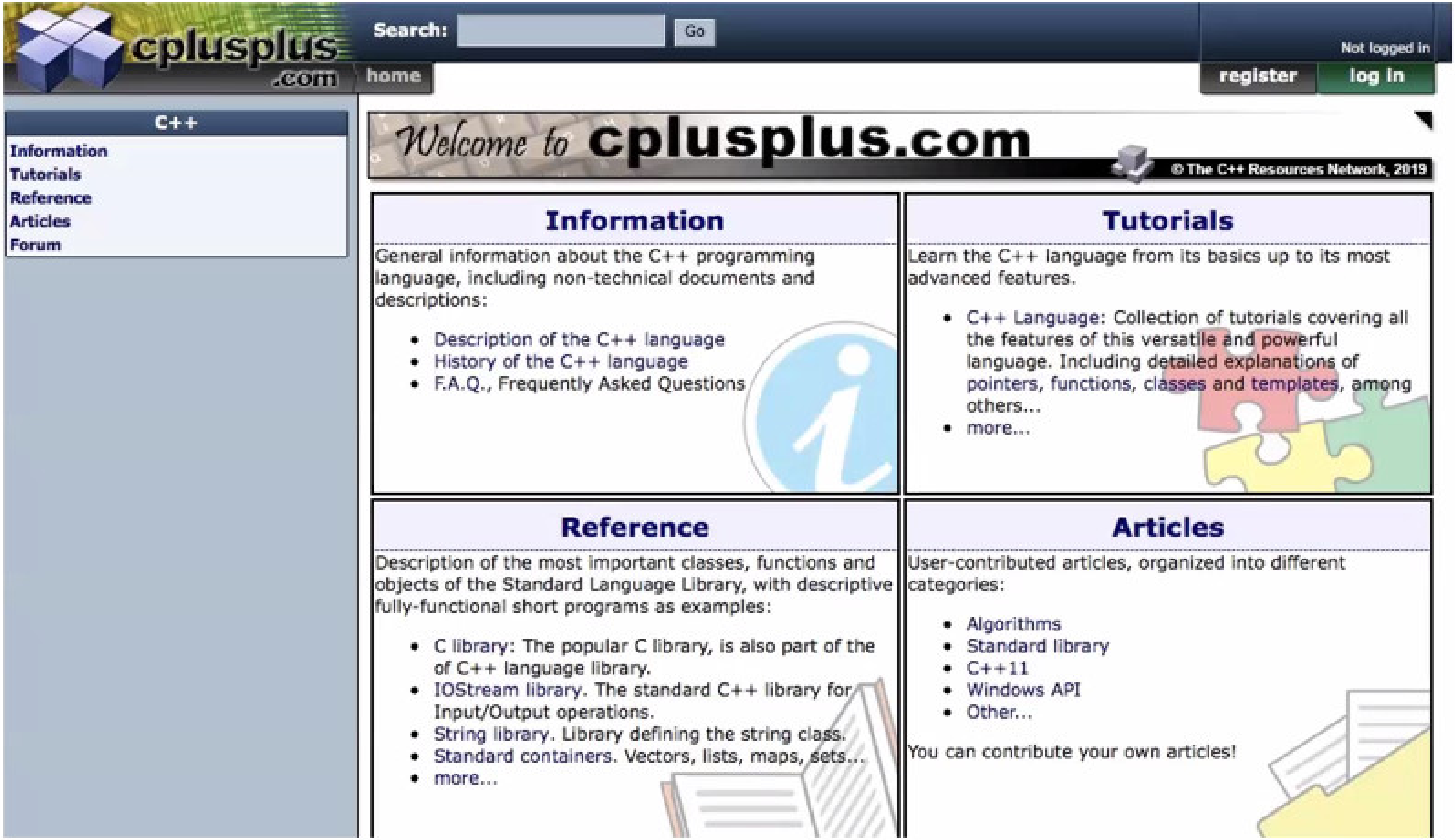
Storica documentazione STL by Silicon Graphics Inc.

`https://www.boost.org/sgi/stl/`

sgi

Standard Template Library Programmer's Guide

Documentazione STL by cplusplus.com



Documentazione STL by cppreference.com

cppreference.com

Create account Search

Page Discussion View View source History

C++ reference

C++98, C++03, C++11, C++14, C++17, C++20

Compiler support (11, 14, 17, 20)
Freestanding implementations

Language
Basic concepts
Keywords
Preprocessor
Expressions
Declaration
Initialization
Functions
Statements
Classes
Templates
Exceptions

Headers

Named requirements

Feature test macros (C++20)

Language support library
Type support — traits (C++11)
Program utilities
Relational comparators (C++20)
numeric_limits — type_info
initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

General utilities library
Smart pointers and allocators
Date and time
Function objects — hash (C++11)
String conversions (C++17)
Utility functions
pair — tuple (C++11)
optional (C++17) — any (C++17)
variant (C++17) — format (C++20)

Strings library
basic_string
basic_string_view (C++17)
Null-terminated strings:
byte — multibyte — wide

Containers library
array (C++11) — vector
map — unordered_map (C++11)
priority_queue — span (C++20)
Other containers:
sequence — associative
unordered associative — adaptors

Iterators library

Ranges library (C++20)

Algorithms library

Numerics library
Common math functions
Mathematical special functions (C++17)
Numeric algorithms
Pseudo-random number generation
Floating-point environment (C++11)
complex — valarray

Localizations library

Input/output library
Stream-based I/O
Synchronized output (C++20)
I/O manipulators

Filesystem library (C++17)

Regular expressions library (C++11)
basic_regex — algorithms

Atomic operations library (C++11)
atomic — atomic_flag
atomic_ref (C++20)

Thread support library (C++11)

Technical specifications

Standard library extensions (library fundamentals TS)
resource_adaptor — invocation_type

Standard library extensions v2 (library fundamentals TS v2)
propagate_const — ostream_joiner — randint
observer_ptr — detection idiom

Standard library extensions v3 (library fundamentals TS v3)
scope_exit — scope_fail — scope_success — unique_resource

Concurrency library extensions (concurrency TS)

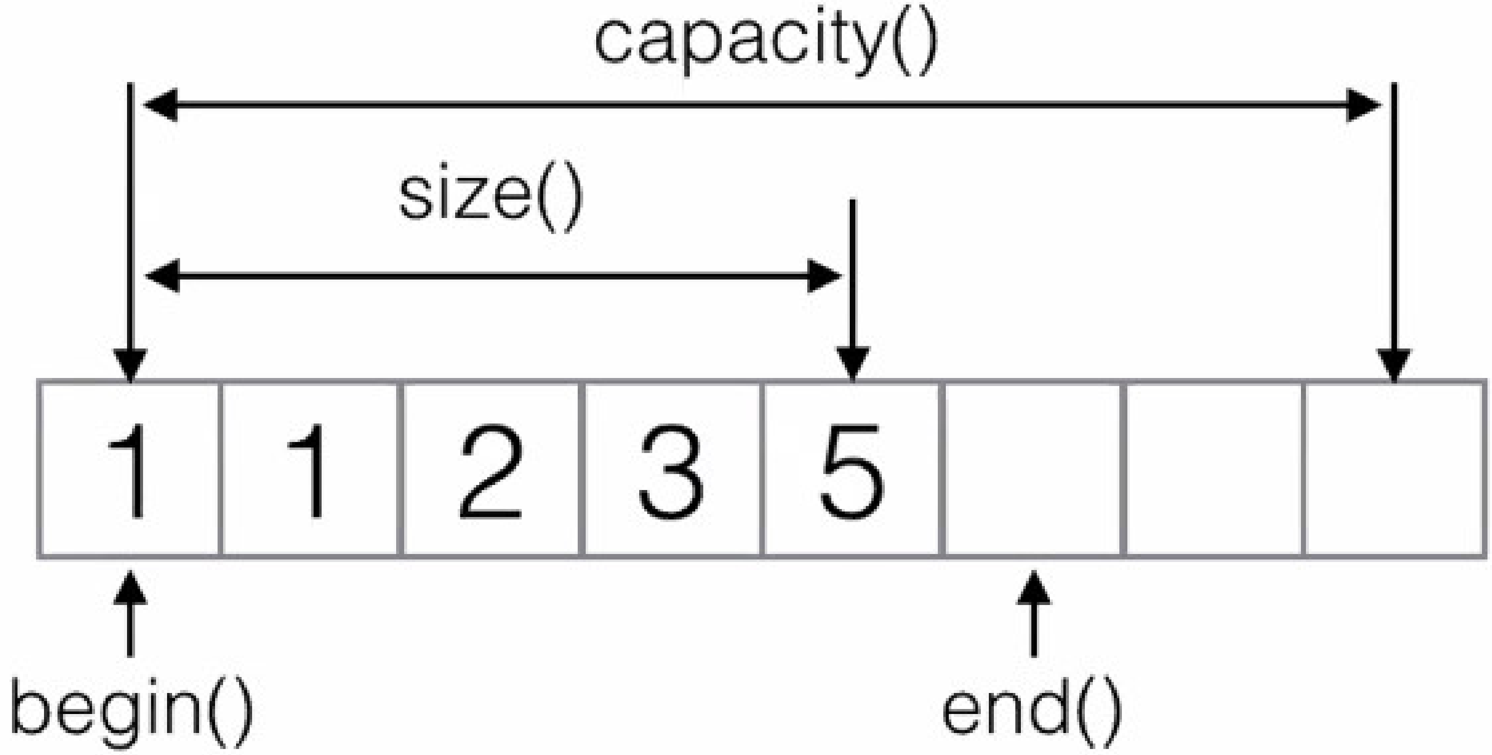
Concepts (concepts TS)

Ranges (ranges TS)

Transactional Memory (TM TS)

External Links — Non-ANSI/ISO Libraries — Index — std Symbol Index

vector



vector è il più semplice contenitore di STL e per la maggioranza delle applicazioni è il più efficiente. È un template di classe che generalizza gli array dinamici.

Caratteristiche di vector:

1) è un contenitore che supporta **l'accesso casuale** agli elementi (accesso in posizione arbitraria in **tempo costante**)

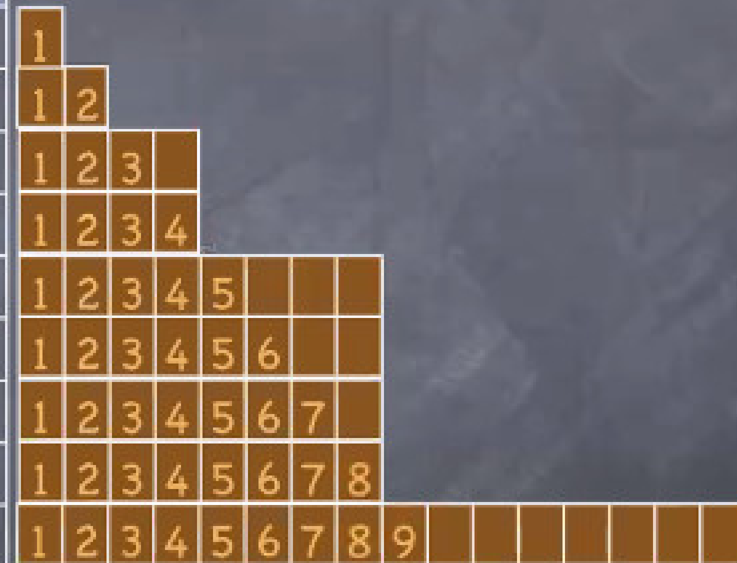
2) inserimento e rimozione in **coda** in **tempo ammortizzato costante**

3) inserimento e rimozione **arbitraria** in **tempo lineare ammortizzato**

4) la **capacità** di un vector può variare dinamicamente

5) la gestione della memoria è **automatica**

Operation	Capacity	Cost
push_back(1)	1	1
push_back(2)	2	1 + 1
push_back(3)	4	2 + 1
push_back(4)	4	1
push_back(5)	8	4 + 1
push_back(6)	8	1
push_back(7)	8	1
push_back(8)	8	1
push_back(9)	16	8 + 1



Cost for the i-th push_back

$$c_i = \begin{cases} 1 + 2^k & \text{if } i - 1 = 2^k \text{ for some } k \\ 1 & \text{otherwise} \end{cases}$$

Thus, n push_back operations cost

$$T(n) = \sum_{i=1}^{n} c_i \leq n + \sum_{i=0}^{\lfloor \lg n \rfloor} 2^i = n + 2n - 1 = 3n - 1.$$

Amortized costs: $T(n)/n = (3n - 1)/n < 3$.

```
template <class _Tp, class _Alloc = __STL_DEFAULT_ALLOCATOR(_Tp) >
class vector : protected _Vector_base<_Tp, _Alloc>
{
...
}
```

`vector` è un template di classe con due parametri di tipo ed un parametro di default per il secondo parametro di tipo.

Ci sono due modi diversi di usare un **`vector`**: lo stile array ereditato dal C e lo stile STL più consono al C++.

```
int a[10];             // array
vector<int> v(10);     // construttore ad 1 argomento
                       // vector(size_type)
                       // con costruzione di default per
                       // gli elementi
```

Si può accedere agli elementi di un **`vector`** con l'operatore di indicizzazione **`operator[]`**

```
int n = 5;
vector <int> v(n);
int a[n] = {2,4,5,2,-2};
for (int i = 0; i < n; i++)
  v[i] = a[i] + 1;
```

```
vector<string> v(10), w;
cout << v.size() << " " << v.capacity(); // 10 10
vector<string> u(v); // costruzione di copia
w = u;               // assegnazione
```

Il metodo `size()` ritorna il numero di elementi contenuti nel vector.

Il metodo `capacity()` ritorna invece la capacità del vector.

invariante: `v.size() <= v.capacity() == true`

```
template <class T>
void stampa(const vector<T>& v) {
  for (int i = 0; i < v.size(); i++)
    cout << v[i] << endl;
}
```

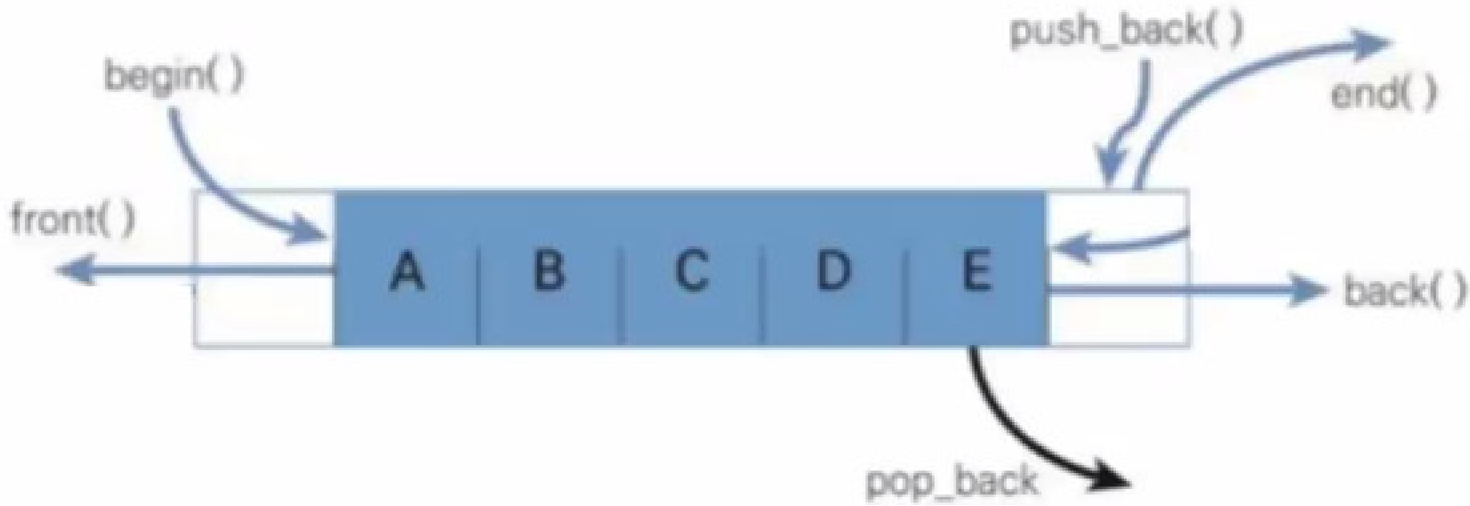
Il costruttore `vector(size_type)` costruisce un vector i cui elementi sono inizializzati con il **costruttore di default**. Il costruttore `vector(size_type n,const T& t)` permette invece di specificare un valore iniziale `t` da cui sono costruiti di copia tutti gli elementi

```
vector <int> ivec(10,-1);
```

In C++03 **non** è possibile inizializzare un `vector` con una data sequenza di valori, diventa invece possibile in C++11.

```
int ia[6] = {-1,5,-7,0,12,3};                 // OK
vector <int> ivec(6) = {-1,5,-7,0,12,3}; // NO C++03
vector <int> ivec(6) = {-1,5,-7,0,12,3}; // OK C++11
```

std::vector




```
void push_back(const T&)
void pop_back()
T& front()
T& back()
iterator begin()
iterator end()
```

Metodi fondamentali

void push_back(const T&): principale metodo di inserimento in (coda ad) un vector, il nuovo elemento inserito è creato con il costruttore di copia

```
void push_back (const value_type& val);
```

Add element at the end

Adds a new element at the end of the vector, after its current last element. The content of *val* is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

```
int main() {
 vector<string> sv; string x;
 while (cin >> x) sv.push_back(x);
 // legge stringhe da cin, separate da spazi, tab o enter
 // fino a end_of_file e le inserisce in coda a sv.
 // Da tastiera end_of_file si invia con una
 // combinazione di tasti particolare:
 // normalmente <Ctrl>+<d>
 cout << endl << "Abbiamo letto:" << endl;
 for (int i = 0; i < sv.size(); i++)
   cout << sv[i] << endl;
}
```

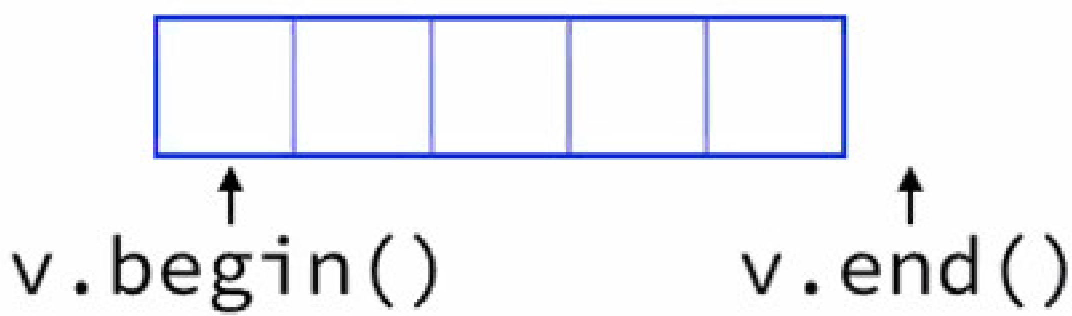
size vs capacity

```
#include <iostream>
#include <vector>

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

for (int i=0; i<100; i++) myvector.push_back(i);

std::cout << "size: " << myvector.size() << '\n';
std::cout << "capacity: " << myvector.capacity() << '\n';
}
```

In modo analogo leggiamo le stringhe di un file separate da spazi, tab o enter e memorizzarle in un vector.

```
int main() {
 vector<string> sv;
 string x;
 ifstream file("dati.txt",ios:in);
 while (file >> x) sv.push_back(x);

 cout << "Abbiamo letto:" << endl;
 vector<string>::iterator it;
 for (it = sv.begin(); it != sv.end(); it++)
   cout << *it << endl;
}
```

`sv.begin()` e `sv.end()` sono degli oggetti della **classe iteratore** su vector.



Ad ogni classe contenitore `C` della STL sono associati due tipi iteratore

`C::iterator`

`C::const_iterator`

Si usa `iterator` quando si necessita un accesso agli elementi del contenitore come lvalue (in lettura e scrittura), se basta un accesso come rvalue (in sola lettura) si preferisce la protezione di `const_iterator`.

Si tratta di cosiddetti ***iteratori bidirezionali***.

```
vector<int>::iterator

vector<int>::const_iterator
```

```
template <class T>
T& vector<T>::iterator::operator*() const;

template <class T>
vector<T>::iterator  vector<T>::begin();
template <class T>
vector<T>::iterator  vector<T>::end();
```

```
template <class T>
const T& vector<T>::const_iterator::operator*() const;

template <class T>
vector<T>::const_iterator  vector<T>::begin() const;
template <class T>
vector<T>::const_iterator  vector<T>::end() const;
```

```
vector<int> v(1); const vector<int> w(2);
iterator it = w.begin();        // ILLEGALE
*(w.begin())=0;                 // ILLEGALE
```

Su ogni tipo iteratore (anche const) di qualche istanza di contenitore `Cont<Tipo>::[const_]iterator` sono **sempre disponibili** le seguenti funzionalità:

```
Cont<Tipo> x;
Cont<Tipo>::[const_]iterator i;

x.begin(); // iteratore che punta al primo elemento
x.end();   // particolare iteratore che non punta ad
           // alcun elemento, e' "un puntatore
           // all'ultimo elemento + 1"
*i;        // elemento puntato da i
i++; ++i;  // puntatore all'elemento successivo. Se
           // i punta all'ultimo elemento di x
           // allora ++i == x.end()
i--; --i;  // puntatore all'elemento precedente. Se
           // i punta al primo elemento di x
           // allora i-- è indefinito (x.begin()-1)
           // (v.end())-- punta all'ultimo elemento
```

Gli iteratori per i contenitori `vector` e `deque` (**contenitori ad accesso casuale**) permettono di avanzare e di retrocedere di un numero arbitrario di elementi in tempo costante. Sono inoltre disponibili gli operatori di confronto per questi iteratori. Si tratta degli ***iteratori ad accesso casuale***.

```
vector<Tipo> v;               // oppure deque<Tipo>
vector<Tipo>::iterator i,j;
int k;

i += k;
i -= k;
j = i+k;
j = i-k;
i < j;
i <= j;
i > j;
i >= j;
```

Tipicamente gli iteratori vengono usati per scorrere gli elementi di un contenitore.

```
Cont<Tipo> x;
...
for(Cont<Tipo>::iterator i = x.begin(); i != x.end(); i++)
{...}
```

I metodi **`empty()`** e **`size()`** sono comuni a tutti i contenitori.

```
x.empty();  // true se x è vuoto, false altrimenti
x.size();   // numero di elementi contenuti in x
```

È possibile inizializzare un `vector` con un segmento di un array o di un `vector` tramite il **template di costruttore**:

```
template<class InputIterator>
vector(InputIterator,InputIterator)
```

incluso escluso

```
int main() {
  int ia[20];
  vector<int> iv(ia,ia+6); // OK
  cout << iv.size() << endl; // size 6
  vector<int> iv2(iv.begin(),iv.end()-2); // OK
  cout << iv2.size() << endl; // size 4
}
```

Metodi di inserimento in un `vector`:

```
string s;
vector<string> vs, vs1;
vector<string>::iterator i;
...
vs.push_back(s); // aggiunge s in coda al vector
vs.insert(i,s);  // aggiunge s subito prima di *i
vs.insert(i,5,s); // aggiunge 5 s subito prima di *i
i = vs.begin() + vs.size()/2;
vs.insert(i, vs1.begin(), vs1.end());
// inserisce tutti gli elementi di vs1 nella
// posizione mediana di vs
// In generale: v.insert(it1,it2,it3) inserisce
// [it2,it3) in v subito prima di it1
```

```
single element (1)  iterator insert (iterator position, const value_type& val);
          fill (2)      void insert (iterator position, size_type n, const value_type& val);
                    template <class InputIterator>
         range (3)      void insert (iterator position, InputIterator first, InputIterator last);
```

```
vs.insert(i,s);
i = vs.begin() + vs.size()/2;
vs.insert(i, vs1.begin(), vs1.end());
```

Attenzione: le operazioni di `insert()` possono risultare piuttosto **inefficienti** e ciò dipende dall'implementazione della classe `vector`

Insert elements

The vector is extended by inserting new elements before the element at the specified *position*, effectively increasing the container size by the number of elements inserted.

This causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

Because vectors use an array as their underlying storage, inserting elements in positions other than the vector end causes the container to relocate all the elements that were after *position* to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).