A Report about Class Vector and Class String

20125071  
Bùi Lê Gia Cát

# Class Vector

A vector is a container similar to a dynamic array, but it can change size while inserting or deleting elements. Vectors' elements are stored in a dynamically allocated array. When adding items, this array may need to be reallocated to expand in size, which requires creating a new array and the relocation of all elements to it. Vectors are not reallocated each time an element is added to the container because it would take too long. Vector containers, on the other hand, may allocate some extra storage to allow for possible growth, resulting in a container with a capacity greater than the storage required to store its elements.

## Constructor

### Default constructor: vector ()

Create an empty vector with no elements. The time complexity is constant.

### Fill constructor: vector (size\_type n, const value\_type val)

Construcs a vector with n element has value *val* if provided.

### Range constructor: vector (iterator\_type first, iterator\_type last)

Constructs a vector with as many elements as the range [*first*, *last*], with each element emplace-constructed in the same order as its range corresponding element.

### Copy constructor: vector (const vector &v)

Create a vector by copying each element in *v* in the same sequence to the new one.

### Move constructor: vector (vector &&v)

Create a container that contains the items of *v*. The time complexity is constant.

### Initializer list constructor: vector (list<value\_type> il)

Create a vector by copying each element in *il* in the same sequence.

## Destructor:

Destroys all elements in the vector and deallocates all allocated storage capacity. The time complexity is linear in the size of the vector.

## Assign operator: vector& operator= (const vector& v)

Assign new contents to the vector, replace existing contents, and resize as needed. The time complexity is linear in size.

## Iterator:

### begin()

Return an iterator pointing to the first element of the vector. The iterator value returned should not be dereferenced if the vector is empty. The time complexity is constant.

### end()

Return an iterator pointing to the end of the vector, after with the vector's last element. This iterator cannot be dereferenced because it does not point to any element. If the vector is not present, the iterator gives the same value as function begin().

### rbegin()

Return a reverse iterator pointing to the vector's last element, just before the one returned by function *end()*. When this type of iterator is increased, it moves to the beginning of the vector.

### rend()

Return a reverse iterator pointing to the theoretical element preceding the vector's first element. The range between *rbegin()* and *rend()* contains all the vector's elements.

### cbegin()

Return a const iterator pointing to the vector's first element. This iterator points to the same element as the one returned by function begin(), but it cannot be used to modify the contents of the vector it points to, even though the vector is not constant. The returning iterator value must not be dereferenced if the container is empty.

### cend()

Return a const iterator pointing directly after the vector's last element. It points to the same location as function *end()*, but it cannot be used to modify the vector's contents, even if the vector is not const. This function returns the same value as cbegin if the vector is empty. The returned value must not be dereferenced.

### crbegin()

Return a const reverse iterator pointing to the vector's last element. Even if the vector is not const, it cannot be used to modify its contents.

### crend ()

Return a const reverse iterator pointing to the theoretical element preceding the vector's first element. Even if the vector is not const, it cannot be used to modify its contents.

## Capacity

### size()

The number of real elements contained in the vector is returned. The time complexity is constant.

### max\_size ()

Return the maximum number of elements that the vector can hold. This is the maximum potential number of elements vector can reach due to the system or library implementation limitations, but it may not reach that size, it can still fail to allocate storage at any point before that size is reached. The time complexity is constant.

### resize (size\_type n, const value\_type &val)

Resize the container to include *n* elements. If n is less than the current vector's size, it is reduced to its first n elements, removing and destroying those beyond. If n is greater than the current vector's size, it appends more elements with the value val if provided until the vector reaches the size of n. If n is greater than the current vector's capacity, the vector is automatically reallocated. The time complexity is linear on the number of construction or destruction of elements. If the vector needs to reallocate, the time complexity is linear in the vector size.

### capacity ()

Returns the size of the vector's current storage space, expressed in terms of elements. This capacity is not always equal to size. It can be equal or greater, with the extra space allowing for expansion without the need to reallocate on each insertion. And the capacity does not impose a limit on the vector's size. And the capacity is not the limit on the size of the vector. The time complexity is constant.

### empty()

Return true if the vector is empty. The time complexity is constant.

### reserve (size\_type n)

Request that the vector capacity is least enough to hold *n* elements. If *n* is greater than the current vector capacity, the vector reallocates its storage to increase its capacity to *n* or greater. Furthermore, this function does not affect the vector's size and cannot change its elements. If the vector must be reallocated, the time complexity is at most linear in vector size.

### shrink\_to\_fit ()

Requests that the vector's capacity be reduced to fit its size. The request is non-binding, and the vector implementation may optimize otherwise, leaving the vector with a capacity greater than its size. This may result in a reallocation, but it does not affect the size of the elements. The time complexity is linear in vector size.

## Element access:

### operator[] (size\_type n)

Return a reference to the vector container element at position *n*. Portable programs should never call this function with an out-of-range argument *n*, as this results in undefined behavior. The time complexity is constant.

### at (size\_type n)

Return a reference to the vector element at position *n*. The function checks whether *n* is within the bounds of valid vector elements and throws an out-of-range exception if it is not. The time complexity is constant.

### front ()

This function, unlike begin(), returns a reference to the first element in the vector rather than the iterator pointing to the first element. When this function is called on an empty vector, the result is undefined. The time complexity is constant.

### back ()

This function returns a reference to the vector's last element, not the iterator pointing to the next element. When this function is called on an empty vector, the result is undefined. The time complexity is constant.

### data ()

Returns a direct pointer to the memory array used by the vector internally to store its owned elements. The time complexity is constant.

## Modifiers

### Assign

Replaces the vector's current contents with new ones and adjusts its size accordingly. If the new vector size exceeds the current vector capacity, the allocated storage space is automatically reallocated.

#### Range version: assign (Input\_Iterator first, Input\_Iterator last)

The new contents are constructed from the elements in the range between first and last, in the same order.

#### Fill version: assign (size\_type n, const value\_type& val)

The new contents are n elements with value val

#### Initializer list: assign (initializer\_list<value\_type> il)

The new elements are copies of the values passed as initializer list, in the same order.

### push\_back (const value\_type& val)

Add an element at the end of the vector. This increases the container size, resulting in an automatic reallocation of the allocated storage space if and only if the new vector size exceeds the current vector capacity.

### pop\_back ()

Remove the last vector element and reduce the container size by one. This destroys the element that was removed.

### Insert

Insert new elements before the element at the specified position, thereby increasing the vector's size. If the new vector size exceeds the current vector capacity, the allocated storage space is automatically reallocated. The time complexity is proportional to the number of inserted elements plus the number of elements after position. There are 5 versions of insert:

* Single element: insert (const\_iterator position, const value\_type& val)
* Fill: insert (const\_iterator position, size\_type n, const value\_type& val)
* Range: insert (const\_iterator position, Input\_Iterator first, Input\_Iterator last)
* Move: insert (const\_iterator position, value\_type&& val)
* Initializer list: insert (const\_iterator position, initializer\_list<value\_type> il)

### Erase

Element removal from the vector This effectively reduces the container size by the number of elements that are removed and destroyed. The time complexity is linear on the number of elements erased and the number of elements after the last element is deleted. There are 2 versions of erase:

* Single element: erase (const\_iterator position)
* Range of elements: erase (const\_iterator first, const\_iterator last)

### swap(vector& v)

Replace the vector's content with the content of another vector object *v* of the same type. The time complexity is constant.

### clear()

Destroy all of the vector's elements and reduce the vector's size to 0. The time complexity is linear in the size of the vector and the time complexity needed to destroy the element.

### emplace (const\_iterator position, Args&&… args)

By inserting a new element at the *position*, the container is extended. This new element is created in place with *args* as the arguments. This effectively doubles the container's size. If the new vector size exceeds the current vector capacity, the allocated storage space is automatically reallocated. The time complexity is linear on the number of elements after position.

### eplace\_back (Args&&… args)

Inserts a new element at the end of the vector, following the vector's current last element. This new element is built in place with args as the constructor arguments. This increases the container size, resulting in an automatic reallocation of the allocated storage space if and only if the new vector size exceeds the current vector capacity. The time complexity is constant.

## Allocator

### get\_allocator()

Return a copy of the allocator object associated with the vector. The time complexity is constant.

# Class String

Strings are objects that represent character sequences. The standard string class supports such objects, with an interface similar to that of a container of bytes but with additional features designed specifically for use with strings of single-byte characters.

## Constructor:

Construct a string, initializing its value depending on the version of the constructor. The time complexity is linear in string length and constant for move constructors.

### Default constructor: string()

Construct an empty string having 0 character.

### Copy constructor: string (const string& str)

Construct a copy of *str.*

### Substring: string (const string& str, size\_t pos, size\_t len = npos)

Copy the portion of *str* that begins at the character position *pos* spans to *len* characters or *npos*.

### From C-string: string (const char\* s)

Copy from the array of characters pointed by *s*.

### From buffer: string (const char\* s, size\_t n)

Copy first *n* characters from the array of characters pointed by *s*.

### Fill: string (size\_t n, char c)

Fill the string with *n* consecutive copies of character *c*.

### Range: string (input\_iterator first, input\_iterator last)

Copy the sequence of characters in the range [*first*, *last*), in the same order.

### Initializer list: string (initializer\_list<char> il)

Copy each of the characters in *il* in the same order.

### Move: string (string&& str)

Acquires the contents of *str*.

## Destructor:

Destroy the string, deallocates all the storage capacity allocated by the string using its allocator. The time complexity is constant.

## Assign operator:

Assigns a new value to the string, replace it's current contents. The time complexity is linear in the new string length. There are 5 versions of assign operator:

* Assign a string: string& operator= (const string& str)
* Assign a C – string: stirng& operator= (const char\* s)
* Assign a character: string& operator= (char c)
* Assign an initializer list: string& operator= (initializer\_list<char> il)
* Move a string: string& operator= (string&& str)

## Iterators:

### begin()

Return an iterator pointing to the first character of the string.

### end()

Return an iterator pointing to the end of the string, starting with the last element of the vector. This iterator cannot be dereferenced because it does not point to any element. If the object is empty, this function returns the same as function begin().

### rbegin()

Return a reverse iterator pointing to the string's final character. When this type of iterator is increased, it moves to the beginning of the string. This function returns an iterator pointing right before the iterator returned by function end().

### rend()

Return a reverse iterator that points to the theoretical element that comes before the first element in the string. The range between rbegin() and rend() contains all of the string's characters.

### cbegin()

Return the same const iterator as function begin(), pointing to the first character of the string. This iterator can be increased and decreased, but it cannot be used to modify the contents it points to, even if the string object is not constant.

### cend()

Return a const iterator pointing directly after the last character of the string, the same as function end(), but it cannot be used to modify the contents, even if the vector is not const. This function returns the same value as function cbegin() if the vector is empty. The returned value must not be dereferenced.

### crbegin()

Return a const reverse iterator pointing to the string's final character. Even if the string is not const, it cannot be used to modify its contents.

### crend ()

Return a const reverse iterator pointing to the theoretical element preceding the string's first character. Even if the string is not const, it cannot be used to modify its contents.

## Capacity:

### size()

The length of the string in bytes is returned. This is the number of actual bytes that confirm the string's contents, which is not always equal to its storage capacity. Size() and length() both return the same value. The time complexity is constant.

### length()

The length() function, like size(), returns the length of the string in bytes. This is the number of actual bytes that confirm the string's contents, which is not always equal to its storage capacity. The complexity of time is constant. The time complexity is constant.

### max\_size()

Return the longest possible length for the string. Due to system or library implementation limitations, this is the maximum potential length of the string, but it may not reach that size, and it may fail to allocate storage at any point before that size is reached. The time complexity is constant.

### resize(size\_t n), resize (size\_t n, char c)

The string is resized to a length of *n* characters. If *n* is less than the current length, the current value is shortened to its first *n* characters, with the characters after the *n*th removed. If *n* is greater than the current length, the current content is extended by inserting as many characters as necessary at the end to reach *n*. If *c* is specified, the new characters are created as duplicates of *c*; otherwise, they are null characters. The time complexity is linear in the new string length.

### capacity()

The size of the storage space currently allocated for the string, expressed in bytes, is returned. This capacity is not always equal to the length of the string. It can be equal to or greater than zero, with the extra space allowing the object to optimize its operations as new characters are added to the string. Furthermore, the capacity is not the string's size limit. The time complexity is constant.

### reserve(size\_t n = 0)

Requests that the capacity be adjusted to adapt a planned size change to a length of up to *n* characters. If n is greater than the container's current capacity, the function causes the container's capacity to be increased to *n* characters (or greater). In all other cases, it is considered a non-binding request to reduce the capacity: the container implementation is free to optimize otherwise and leave the string with a capacity greater than *n*. This function does not affect the length and cannot change the content. The time complexity is constant.

### clear()

Delete all the characters in the string and the string itself become empty. The time complexity is constant.

### empty()

Return whether the string is empty. The time complexity is constant.

### shrink\_to\_fit()

Requests that the string's capacity be reduced to fit its size. The request is non-binding, and the container implementation is free to optimize otherwise, leaving the string with a capacity larger than its size. This function does not affect the length and cannot change the content. The time complexity is constant.

## Element access:

### operator[] (size\_t pos)

The character at position *pos* in the string is returned as a reference. If *pos* equals string length, the function returns a pointer to the null character that comes after the last character in the string. The time complexity is constant.

### at(size\_t pos)

The character at position *pos* in the string is returned as a reference. The function checks whether *pos* is a valid position of a character in the string and throws an out-of-range exception if it is not. The time complexity is constant.

### front()

Unlike the function *begin()*, which returns an iterator pointing to the first character, this function returns a reference to the first character of the string. The time complexity is constant.

### back()

Return a reference to the last character of the string. The time complexity is constant.

## Modifiers:

### Operator+=

Add a character or a sequence of characters at the end of the string. The time complexity is linear in the new string length. There are 4 versions of this operator:

* String: string& operator+= (const string& str)
* C-string: string& operator+= (const char\* s)
* Character: string& operator+= (char c)
* Initializer list: string& operator+= (initializer\_list<char> il)

### Append

Add a character or a sequence of characters at the end of the string. The time complexity is linear in the new string length.

#### String: string& append (const string& str)

Append a copy of *str*

#### Substring: string& append (const string& str, size\_t subpos, size\_t sublen)

Append a copy of a substring of *str*. The substring is the portion of str that begins at the character position *subpos* and spans *sublen* characters.

#### C-string: string& append (const char\* s)

Append a copy of the C-string pointed by *s*.

#### Buffer: string& append (const char\* s, size\_t n)

Append a copy of the first *n* characters of the C-string pointed by *s*.

#### Fill: string& append (size\_t n, char c)

Append *n* consecutive copies of character *c*.

#### Range: string& append (input\_iterator first, input\_iterator last)

Append a copy of the sequence of characters in the range [*first*, *last*), in the same order.

#### Initializer list: string& append (initializer\_list<char> il)

Append a copy of each of the characters in *il*, in the same order.

### push\_back(char c)

Add character *c* to the end of the string. The time complexity is linear in the new string length.

### Assign

Assign a new value to the string, replacing its current contents. The time complexity is linear in the new string length and constant for the move version.

#### String: string& assign (const string& str)

Copy *str* to the current string.

#### Substring: string& assign (const string& str, size\_t subpos, size\_t sublen)

Copies the portion of *str* that begins at the character position *subpos* and spans *sublen* characters.

#### C-string: string& assign (const char\* s)

Copies the C-string pointed by *s*.

#### Buffer: string& assign (const char\* s, size\_t n)

Copies the first *n* characters from the C-string pointed by *s*.

#### Fill: string& assign (size\_t n, char c)

Replace the current string with *n* consecutive copies of character *c*.

#### Range: string& assign (input\_iterator first, input\_iterator last)

Copy the sequence of characters in the range [*first*, *last*), in the same order.

#### Initializer list: string& assign (initializer\_list<char> il)

Copy each of the characters in *il*, in the same order.

#### Move: string& assign(string&& str)

Acquires the contents of *str*.

### Insert

Insert a character or a sequence of characters into the string right before position *pos*.

#### String: string& insert (size\_t pos, const string& str)

Insert a copy of *str*.

#### Substring: string& insert (size\_t pos, const string& str, size\_t subpos, size\_t sublen)

Inserts a copy of a substring of *str*. The substring is the portion of *str* that begins at the character position *subpos* and spans *sublen* characters.

#### C-string: string& insert (size\_t pos, const char\* s)

Insert a copy of the string formed by C-string pointed *s*.

#### Buffer: string& insert (size\_t pos, const char\* s, size\_t n)

Insert a copy of the first *n* characters in the C-string pointed by *s*.

#### Fill: string& insert (size\_t pos, size\_t n, char c), iterator insert (const\_iterator p, size\_t n, char c)

Inserts *n* consecutive copies of character *c*.

#### Single character: iterator insert (const\_iterator p, char c)

Inserts character *c*.

#### Range: iterator insert (iterator p, input\_iterator first, input\_iterator last)

Inserts a copy of the sequence of characters in the range [*first*, *last*), in the same order.

#### Initializer list: string& insert (const\_iterator p, initializer\_list<char> il)

Inserts a copy of each of the characters in *il*, in the same order.

### Erase

Erase a part of the string. The time complexity is linear in the new string length.

#### Sequence: string& erase (size\_t pos=0, size\_t len = npos)

Erases the portion of the string value that begins at the character position *pos* and spans *len* characters. The default argument erases all characters in the string like function *clear()*.

#### Character: iterator erase (const\_iterator p)

Erases the character pointed by *p*.

#### Range: iterator erase (const\_iterator first, const\_iterator last)

Erases the sequence of characters in the range [*first*, *last*).

### Replace

Replace the portion of the string that begins at character *pos* and spans *len* characters. The time complexity is linear in the new string length.

#### String: string& replace (size\_t pos, size\_t len, const string& str), string& replace (const\_iterator i1, const\_iterator i2, const string& str)

Copy *str*.

#### Substring: string& replace (size\_t pos, size\_t len, const string& str, size\_t subpos, size\_t sublen)

Copy the portion of *str* that begins at the character position *subpos* and spans *sublen* characters.

#### C-string: string& replace (size\_t pos, size\_t len, const char\* s), string& replace (const\_iterator i1, const\_iterator i2, const char\* s)

Copy C-string pointed by *s*.

#### Buffer: string& replace (size\_t pos, size\_t len, const char\* s, size\_t n), string& replace (const\_iterator i1, const\_iterator i2, const char\* s, size\_t n)

Copies the first *n* characters from the C-string pointed by *s*.

#### Fill: string& replace (size\_t pos, size\_t len, size\_t n, char c), string& replace (const\_iterator i1, const\_iterator i2, size\_t n, char c)

Replaces the portion of the string by *n* consecutive copies of character *c*.

#### Range: string& replace (const\_iterator i1, const\_iterator i2, input\_iterator first, input\_iterator last)

Copies the sequence of characters in the range [*first*, *last*), in the same order.

#### Initializer list: string& replace (const\_iterator i1, const\_iterator i2, initializer\_list<char> il)

Copies each of the characters in *il*, in the same order.

### swap(string& str)

Exchange the content of the container by the content of *str*. The time complexity is constant.

### pop\_back()

Delete the last character of the string, reducing its length by one. The time complexity is constant.

## String operations:

### c\_str()

Return a pointer to a C-string representing the current value of the string object and an additional null character at the end. The time complexity is constant.

### data()

Return a pointer to a C-string representing the current value of the string object and an additional null character at the end. The time complexity is constant.

### get\_allocator()

Returns a copy of the allocator object associated with the string. The time complexity is constant.

### copy (char \*s, size\_t len, size\_t pos = 0)

Copies a substring of the string object's current value into the array pointed by *s*. This substring is made up of the len characters that begin at position *pos*. The function does not append a null character to the end of the copied data. The time complexity is linear in the number of characters copied.

### Find

Look for the first occurrence of the sequence specified by its arguments in the string. Unlike the function find\_first\_of(), when searching for more than one character, it is not enough for just one of these characters to match; the entire sequence must match. The time complexity is linear in the length of the sequence to match in the worst case. There are 4 versions of find:

* String: size\_t find (const string& str, size\_t pos = 0)
* C-string: size\_t find (const char\* s, size\_t pos = 0)
* Buffer: size\_t find (const char\* s, size\_t pos, size\_type n)
* Character: size\_t find (char c, size\_t pos = 0)

### rfind

Look for the last occurrence of the sequence specified by its arguments in the string. When pos is specified, the search includes only character sequences that begin at or before position pos, ignoring any possible match that begins after pos. The time complexity is linear in the string length multiply the number of characters to match in the worst case. There are 4 versions of rfind:

* String: size\_t rfind (const string& str, size\_t pos = npos)
* C-string: size\_t rfind (const char\* s, size\_t pos = npos)
* Buffer: size\_t rfind (const char\* s, size\_t pos, size\_t n)
* Character: size\_t rfind (char c, size\_t pos = npos)

### Find\_first\_of

Search the string for the first character that matches any of the characters specified in its arguments. When *pos* is specified, the search only includes characters that occur at or after position *pos*, ignoring any possible occurrences before *pos*. It is sufficient for a single character in the sequence to match, rather than all of them, as in function *find()*. The time complexity is linear in the length of the string multiply the number of characters to match in the worst case. There are 4 versions of find\_first\_of:

* String: size\_t find\_first\_of (const string& str, size\_t pos = 0)
* C-string: size\_t find\_first\_of (const char\* s, size\_t pos = 0)
* Buffer: size\_t find\_first\_of (const char\* s, size\_t pos, size\_t n)
* Character: size\_t find\_first\_of (char c, size\_t pos = 0)

### Find\_last\_of

Search the string for the last character that matches any of the characters specified in its arguments. When *pos* is specified, the search only includes characters that are at or before position pos, ignoring any potential occurrences after *pos*. The time complexity is linear in the string length multiply the number of characters to match in the worst case. There are 4 versions of find\_last\_of:

* String: size\_t find\_last\_of (const string& str, size\_t pos = npos)
* C-string: size\_t find\_last\_of (const char\* s, size\_t pos = npos)
* Buffer: size\_t find\_last\_of (const char\* s, size\_t pos, size\_t n)
* Character: size\_t find\_last\_of (char c, size\_t pos = npo)

### Find\_first\_not\_of

Search the string for the first character that does not match any of the characters specified in its arguments. When *pos* is specified, the search only includes characters that occur at or after position *pos*, ignoring any possible occurrences before that character. The time complexity is linear in string length multiply the number of characters to match in the worst case. There are 4 versions of find\_first\_not\_of

* String: size\_t find\_first\_not\_of (const string& str, size\_t pos = 0)
* C-string: size\_t find\_first\_not\_of (const char\* s, size\_t pos = 0)
* Buffer: size\_t find\_first\_not\_of (const char\* s, size\_t pos, size\_t n)
* Character: size\_t find\_first\_not\_of (char c, size\_t pos = 0)

### Find\_last\_not\_of

Searches the string for the last character that does not match any of the characters specified in its arguments. When *pos* is specified, the search only includes characters that are at or before position *pos*, ignoring any potential occurrences after *pos*. The time complexity is linear in string length multiply the number of characters to match in the worst case. There are 4 versions of find\_last\_not\_of:

* String: size\_t find\_last\_not\_of (const string& str, size\_t pos = npos)
* C-string: size\_t find\_last\_not\_of (const char\* s, size\_t pos = npos)
* Buffer: size\_t find\_last\_not\_of (const char\* s, size\_t pos = npos)
* Character: size\_t find\_last\_not\_of (char c, size\_t pos = npos)

### substr(size\_t pos = 0, size\_t len = npos)

Returns a newly constructed string object with its value set to a copy of this object's substring. The substring is the part of the object that begins at character position *pos* and spans *len* characters or until the string ends. The time complexity is linear in the length of the returned string.

### Compare

Compares the string object's or a substring's value to the sequence of characters specified by its arguments. The time complexity is linear in both the compared and comparing string lengths. There are 4 versions of compare:

* String: int compare (const string& str)
* Substring: int compare (size\_t pos, size\_t len, const string& str),  
   int compare (size\_t pos, size\_t len, const string& str, size\_t subpos,  
   size\_t sublen)
* C-string: int compare (const char\* s),  
   int compare (size\_t pos, size\_t len, const char\* s)
* Buffer: int compare (size\_t pos, size\_t len, const char\* s, size\_t n)

## Member constants:

### npos

A static member constant value containing the maximum possible value for an element. With a value of 1, this constant is defined. When used as a length parameter value, it means "until the end of the string."

# Reference

https://www.cplusplus.com/reference/vector/vector/

https://www.cplusplus.com/reference/string/string/