Advanced STL

Contents

[01 Learn STL - Member Functions vs Algorithm Functions 2](#_Toc392520222)

[02 Learn STL - Tricky Reverse Iterator 4](#_Toc392520223)

[03 Learn STL - Equivalence vs Equality 6](#_Toc392520224)

[04 Learn STL - Removing Elements 1 7](#_Toc392520225)

[05 Learn STL - Removing Elements 2 8](#_Toc392520226)

[06 Learn STL - Vector vs. Deque - part I 9](#_Toc392520227)

[07 Learn STL - Vector vs. Deque - part II – Combined 9](#_Toc392520228)

[08 Learn STL - Object Slicing 13](#_Toc392520229)

# 01 Learn STL - Member Functions vs Algorithm Functions

#include <iostream>

#include <unordered\_set>

#include <map>

#include <list>

#include <algorithm>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Containers Member Functions vs Algorithm Functions

Containers Member Functions: Functions that work on containers

data.

Algorithm: Also functions that work on containers data.

The question is could there be any duplicate functions between these

two groups?

Answer: YES

List:

List Member Function Algorithms

void remove(const T) template<class Comp> void remove\_if(Comp);

void unique(); template<class Comp> void unique(Comp);

Void sort(); template<class Comp> void sort(Comp);

void merge(list&); template<class Comp> void merge(Comp);

void reverse();

Associative Containers:

size\_type count(const T&) const;

iterator find(const T&) const;

iterator lower\_bound(const T&) const;

iterator upper\_bound(const T&) const;

pair<iterator, iterator> equal\_range(const T&) const;

Unordered Container:

size\_type count(const T&) const;

iterator find(const T&) const;

pair<iterator, iterator> equal\_range(const T&) const;

No the question is why there are duplicate function?

- Although there are duplicate functions and have the same purpose,

they are actually different. They do things in different way. So,

now again the question is which one is better?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

{

unordered\_set<int> s = { 2, 4, 1, 8, 5, 9 };

unordered\_set<int>::iterator itr;

// Using member function

itr = s.find(4); // O(1)

// Using algorithm // O(n)

// Why algorithm takes O(n)? This is because, algorithm find()

// Only knows data through iterator. It doesn't know anything

// about container. It doesn't know whether the data is store in

// hash table, or in a binary tree or in contiguous array.

// The only way to find the element is starting from first iterator

// and iterate through all the items till the end and find the data.

// This takes linear time.

itr = find(s.begin(), s.end(), 4);

}

{

map<char, string> myMap = { { 'S', "Sunday" }, { 'M', "Monday" }, { 'T', "Tuesday" } };

// Using member function -  O(log(n))

map<char, string>::iterator itr;

itr = myMap.find('F');

// Using Algorithm - O(n)

// Algorithm find() is much complex, cos, it need two things to

// find. 'F' and "Friday" and the compare function will be complex.

itr = find(myMap.begin(), myMap.end(), make\_pair('F', "Friday"));

}

{

list<int> s = { 2, 4, 1, 8, 5, 9 };

// Since list contains unsorted data, the only way to find and

// remove the item is to iterate through each item.

// Using member function - O(n)

// Member function remove() just tweaks two pointers

/\*

+---+--->+---+--->+---+--->+---+--->+---+--->+---+

| 2 |    | 4 |    | 1 |    | 8 |    | 5 |    | 9 |

+---+<---+---+<---+---+<---+---+<---+---+<---+---+

 +-----+---->

 |  4  |

<----+-----+

+-----+-----------------> +-----+----> +-----+---> +-----+----> +-----+

|  2  |                  |  1  |      |  8  |     |  5  |      |  9  |

+-----+ <-----------------+-----+ <----+-----+ <---+-----+ <----+-----+

\*/

s.remove(4); //s = { 2, 1, 8, 5, 9 };

list<int>::iterator itr;

// Using Algorithm - O(n)

// Algorithm removes the elements by copying and after the removal

// the content of s will be { 2, 1, 8, 5, 9, 9 }. It has one more

// 9 at the end. This is because, the algorithm don't know about

// the container and it can't remove the element. Then how to remove

// the another 9? remove() returns the iterator to last element.

itr = remove(s.begin(), s.end(), 4); //s = { 2, 1, 8, 5, 9, 9 };

s.erase(itr, s.end());//s = { 2, 1, 8, 5, 9 };

/\*

+-----+----> +-----+----> +-----+----> +-----+---> +-----+----> +-----+

|  2  |      |  1  |      |  8  |      |  5  |     |  9  |      |  9  |

+-----+ <----+-----+ <----+-----+ <----+-----+ <---+-----+ <----+-----+

\*/

//To sort, with member functions...

s.sort();

// To sort with algorithm,

// Below code doesn't work. Because, sort() needs random access

// iterators. List has bi-directional iterators.

//sort(s.begin(), s.end());

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary:

1. There are duplicated function between container's member function

and algorithm functions.

2. Prefer member functions over algorithm function with the same name

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

return 0;

}

# 02 Learn STL - Tricky Reverse Iterator

#include <iostream>

#include <vector>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Reverse Iterator

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(void)

{

// Two ways to declare reverse iterator

reverse\_iterator<vector<int>::iterator> ritr1;

vector<int>::reverse\_iterator ritr2;

// Traversing with reverse iterator

vector<int> vec = { 4, 5, 6, 7 };

for (ritr1 = vec.rbegin(); ritr1 != vec.rend(); ++ritr1)

cout << \*ritr1 << " ";

cout << endl;

// Convention

vector<int>::iterator itr;

vector<int>::reverse\_iterator ritr;

// Converting regular iterator to reverse iterator

ritr = vector<int>::reverse\_iterator(itr);

// Converting the reverse iterator to iterator is not possible

// as shown above. We need to use the reverse iterator member

// function base()

//itr = vector<int>::iterator(ritr);

itr = ritr.base();

// BUT THERE IS A DIFFERENCE

vector<int> aVec = { 1, 2, 3, 4, 5 };

vector<int>::reverse\_iterator aRitr = find(aVec.rbegin(), aVec.rend(), 3);

cout << "Using Reversion Iterator: " << \*aRitr << endl;

cout << "Converting Reverse iterator to iterator!!!" << endl;

vector<int>::iterator anotherItr = aRitr.base();

cout << "Using Iterator after conversion: " << \*anotherItr << endl;

/\*OUTPUT

Using Reversion Iterator: 3

Converting Reverse iterator to iterator!!!

Using Iterator after conversion: 4

----------------

This is because of below...

vec.begin()     itr = ritr.base()  vec.end()

|                |                  |

|                |                  |

|                |                  |

v  v v

   +----------------------------------+

   |            | n |n+1|             |

   +----------------------------------+

 ^               ^                   ^

 |               |                   |

 |               |                   |

 |               |                   |

vec.rend()         ritr             vec.rbegin()

\*/

// Inserting elements

//aVec.insert(aRitr.base(), 9);

//aVec.insert(aRitr, 9);

return 0;

}

# 03 Learn STL - Equivalence vs Equality

class Lsb\_less

{

public:

bool operator()(int x, int y)

{

return ((x % 10) < (y % 10));

}

};

int main(void)

{

set<int, Lsb\_less> s = { 21, 23, 26, 27 };

set<int, Lsb\_less>::iterator itr1, itr2;

// itr1 will be pointing to s.end().

// Algorithm function will return the end

itr1 = find(s.begin(), s.end(), 36);

// itr2 will be pointing 26!!!

itr2 = s.find(36);

// This is because, the Algorithm find uses if(x == y)

// The member function uses if( !(x<y) && !(y<x) )

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

If the function is using operator "<" or its like, it's checking equivalence

- Typically it's algorithm that works with sorted data, or a member

function of a container with sorted data, such as associative container.

- If the function is using operator"==" or its like, it's checking equality

- Typically the data is not required to be sorted

Algorithms of equality

search();

find\_end();

find\_first\_of();

adjacent\_search();

Algorithms of equivalence:

binary\_search();

includes();

lower\_bound();

upper\_bound();

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 04 Learn STL - Removing Elements 1

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

using namespace std;

// Prints contents of a container!!!

template<typename T>

void print(T t, string msg)

{

cout << msg << " { ";

for (typename T::iterator it = t.begin(); it != t.end(); ++it)

cout << \*it << ", ";

cout << " }" << endl;

}

int main(void)

{

vector<int> c = { 1, 4, 6, 1, 1, 1, 1, 12, 18, 16 };

print(c, "Original: ");

//for (auto itr = c.begin(); itr != c.end();)

//{

// if (\*itr == 1)

// {

// itr = c.erase(itr);

// }

// else

// {

// itr++;

// }

//} // Complexity: O(n\*m)

//print(c, "After erase():");

auto itr = remove(c.begin(), c.end(), 1);// Complexity is O(n)

c.erase(itr, c.end());

print(c, "After erase():");

cout << "Capacity is :" << c.capacity() << endl;

//c.shrink\_to\_fit(); // shrink\_to\_fit() is present in C++11

// We can use swap trick

vector<int>(c).swap(c);

cout << "Capacity is :" << c.capacity() << endl;

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary of removing elements:

1. Vector or deque: algorithm remove() followed by erase()

2. List: Member function remove()

3. Associative container or unordered container: erase()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 05 Learn STL - Removing Elements 2

int main(void)

{

{

multiset<int> c = { 1, 4, 6, 1, 1, 1, 1, 12, 18, 16 };

cout << "###################### Multiset ######################" << endl;

//for (multiset<int>::iterator itr = c.begin(); itr != c.end(); ++itr)

//{

// if (\*itr == 1)

// {

// c.erase(itr);

// cout << "Erased one item of " << \*itr << endl;

// }

//}

for (multiset<int>::iterator itr = c.begin(); itr != c.end(); ++itr)

{

if (\*itr == 1)

{

cout << "Erased one item of " << \*itr << endl;

c.erase(itr++);

}

else

{

itr++;

}

}

}

{

cout << "###################### Vector ######################" << endl;

vector<int> c = { 1, 4, 6, 1, 1, 1, 1, 12, 18, 16 };

for (auto itr = c.begin(); itr != c.end(); ++itr)

{

if (\*itr == 1)

{

cout << "Erased one item of " << \*itr << endl;

itr = c.erase(itr);

}

else

{

itr++;

}

}

}

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Summary:

Sequence container and unordered container: itr = c.erase(itr);

Associative container: c.erase(itr++);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 06 Learn STL - Vector vs. Deque - part I

# 07 Learn STL - Vector vs. Deque - part II – Combined

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Vector vs. Deque

Vector can grow at only one end.

Deque can grow at both the ends.

  +---------------------------+

  |          VECTOR           ------->

  +---------------------------+

  +---------------------------+

  <-------            DEQUE            -------->

  +---------------------------+

Which one to use?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class Dog{ };

int main(void)

{

#pragma region Vector

{

vector<int> vec = { 2, 3, 4, 5 };

cout << "Size of vec: " << vec.size() << endl;

cout << "Capacity of vec: " << vec.capacity() << endl;

vec.push\_back(6);

cout << "Size of vec: " << vec.size() << endl;

cout << "Capacity of vec: " << vec.capacity() << endl;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

OUTPUT

Size of vec: 4

Capacity of vec: 4

Size of vec: 5

Capacity of vec: 6 // Capacity is more here

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Why the capacity is more in the later case?

+---+---+---+---+

| 2 | 3 | 4 | 5 | here vec.size() == vec.capacity()

+---+---+---+---+

vec.push\_back(6); Now there is no space to push the data into the

vector. It creates new vector which is 1.5 capacity of original.

The old vector is deleted and space is released and involves the

process of reallocation.

+---+---+---+---+---+---+

| 2 | 3 | 4 | 5 | 6 |   |

+---+---+---+---+---+---+

VECTOR CAPACITY GROWS EXPONENTIALLY!!!

DRAWBACKS:

1. Expensive reallocation.

2. Requires contiguous memory.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

{

// 6 Dogs are created with default constructor!!!

vector<Dog> vec(6);

cout << "Size of vec: " << vec.size() << endl;

cout << "Capacity of vec: " << vec.capacity() << endl;

// vec.size() == 0 and vec.capacity() == 0

vector<Dog> vec2;

vec2.resize(6); // 6 Dogs are created with default constructor!!!

cout << "Size of vec2: " << vec.size() << endl;

cout << "Capacity of vec2: " << vec.capacity() << endl;

vector<Dog> vec3;

// reserve() function increase the capacity but not the size.

vec3.reserve(6); // No Dog's are constructed

cout << "Size of vec2: " << vec.size() << endl;

cout << "Capacity of vec2: " << vec.capacity() << endl;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Strategy of minimizing reallocation:

1. If the maximum number of item is known, use reserve(MAX);

2. If the end of grow is know, reserve as much memory as we can.

Once all data is inserted, trim off the rest of data.

To trim off,

vec.shrink\_to\_fit(); // C++11

vector<int>(vec).swap(vec); // C++03

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Vector

#pragma region Deque

/\*

  +---------------------------+

  <-------            DEQUE            -------->

  +---------------------------+

Underlying implementation of Deque

- A deque starts with fixed sized array to hold the data and when the

array is full and there is no more space for new elements to PUSH\_BACK(),

It allocates another fix sized array after it...

- If there is no more space left to PUSH\_FRONT(), it allocates another

fix sized array before it...

+---+   +---+   +---+

|   |   |   |   |   |

+---+   +---+   |   |

|   |   |   |   | ^ |

+---+   +---+   | | |

|   |   |   |   | | |

   <---- +---+   +---+   |   | ---->

|   |   |   |   |   |

|   |   +---+   +---+

| | |   |   |   |   |

| | |   +---+   +---+

| v |   |   |   |   |

|   |   +---+   +---+

|   |   |   |   |   |

+---+   +---+   +---+

- Deque grows linearly with a fixed size...

OBSERVATION:

- No reallocation.

deque has no reserve() and capacity()

- Slightly slower than vector because

1. More complex data structure.

2. Locality (Not in contiguous memory, more page faults)

\*/

#pragma endregion Deque

#pragma region Which one to use

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Deque or Vector, which one to use?

- Need to push\_front() a lot?-> deque is the answer.

- Performance is important? -> vector is the answer.

Still lot more things needs to be considered...

Note 1: Element Type

- When the elements are not of trivial types(Built in data types),

deque is not much less efficient than vector.

Note 2: Memory availability

- Allocation of large contiguous memory is a problem? -> deque

cos, vector requires large contiguous amount of memory.

Note 3: Frequency of unpredictable growth.

Example:

vector<int> vec;

for (int x = 0; x < 1025; x++)

vec.push\_back(x); // To push\_back 1024 elements, there

will be a minimum of 11 REALLOCATION.

- We know REALLOCATION is costly. To prevent this we can use

vec.reserve(1024);

- If the frequency of growth is unpredictable, ->deque

Note 4: Invalidation of pointers/references/iterators because

of growth.

##############################

vector<int> vec = { 2, 3, 4, 5 };

int\* p = &vec[3];

vec.push\_back(6);

cout << \*p << endl; // UNDEFINED BEHAVIOR!!!

- The push\_back(6) invalidates all the pointer and iterators.

- Hence the above code is UNDEFINED BEHAVIOR!!!

deque<int> deq = { 2, 3, 4, 5 };

p = &deq[3];

deq.push\_back(6);

cout << \*p << endl; // OK : 5

- For deque, if it is growing at both ends, all the pointers will

not be invalidated.

- But if we insert/remove in the middle of the deque, the pointers

or references or iterators will be invalidated!!!

##############################

Note 5: Vector's unique function : Portal to C language

Example:

vector<int> vec = { 2, 3, 4, 5 };

void cFunction(const int\* arr, int size);

cFunction(&vec[0], vec.size());

// Passing data from a list to C

list<int> myList;

// Copy the list data to vector and call the cFunction()

vector<int> vec(myList.begin(), myList.end());

cFunction(&vec[0], vec.size());

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#pragma endregion Which one to use

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SUMMARY

1. Frequent push\_front() - deque

2. Build-in data type - vector

3. Not build-in data type - deque

4. Contiguous memory - deque

5. Unpredictable growth - deque

6. Pointer integrity - deque

7. Frequently passed to C  - vector

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# 08 Learn STL - Object Slicing

#include <iostream>

#include <string>

#include <deque>

using namespace std;

class Dog

{

public:

virtual void bark() { cout << "I don't have a name." << endl; }

};

class YellowDog : public Dog

{

string name\_;

public:

YellowDog(string name) : name\_(name) { }

void bark() { cout << "My name is: " << name\_ << endl; }

};

int main(void)

{

#pragma region Object Slicing

{

deque<Dog> d;

YellowDog y("Gunner");

d.push\_front(y);

d[0].bark();

}

#pragma endregion Object Slicing

{

deque<Dog\*> d;

YellowDog y("Gunner");

d.push\_front(&y);

d[0]->bark();

}

return 0;

}