Object Oriented Programming with C++

16. Standard Template Library (STL)

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

What is STL

- STL is a collection of C++ template classes (containers) and template functions (algorithms) which provide support for common programming data structure (such as array, list, set, map, stack, queue etc...) and common algorithms (such as sort, reverse, copy etc...).

Core components of STL

- Containers
  - implemented as class templates for different kind of data structures (e.g. Array, set, map, stack, queue)
- Algorithms
  - implemented as function templates – they are not friend function or methods of containers
  - algorithms do not directly access elements present in containers, they access elements in containers through iterators
- Iterators
  - iterators are type members of the container class templates
  - instances of iterators are pointer-like object which can be incremented with ++, dereferenced with *, and compared against another iterator with !=
    - iterators are classified according to operations that they support
  - Iterator can be used to iterate over elements stored in containers
  - every container provides iterator (implementation differs across containers)

STL Sequence Containers

- Sequence containers
  - vector
    - A contiguously allocated sequence of T
    - The default choice of container
  - list
    - Doubly linked list of T
    - Use when you need to insert or delete element without moving existing elements
  - forward_list
    - A singly linked list of T
    - Use only if you are planning to store very few elements
  - deque
    - A double ended queue of T
    - Memory might not be contiguous (its implementation specific)

STL Sequence Containers

- Sequence containers
  - vector (https://www.geeksforgeeks.org/vector-in-cpp-stl/)
    - #include<vector>
    - Allocates memory in chunks. So ***size*** and ***capacity*** may differ
    - Allows random access of elements in constant time using [ ] and at() method
      - vector class template overloads [ ]
      - [ ] doesn’t check for array bounds, while at() does check array bounds
      - **int x; vector<int> v;..... x = v[i];** and **v[i] = x;** result in undefined behaviour if **i>v.size-1**
      - v.at(i) method throws exception if **i>v.size-1**
    - Adding/removing element from back takes constant time (Except when it needs to reallocate all the elements)
    - Adding/removing elements from front or middle is costly (O(n)) compared to adding/removing elements from back

```
int main()
{
    std::vector<int> int_vect;

    for(int i = 0; i < 10; i++) {
        int_vect.push_back(i+1);
        cout << i << " ";
        cout << int_vect.size() << " ";
        cout << int_vect.capacity() << endl;
    }

    for(int i = 0; i < 10; i++) {
        cout << int_vect[i] << " ";
    }
    cout << endl;
    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = int_vect.cbegin(); it != int_vect.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = int_vect.crbegin(); it != int_vect.crend(); ++it)
        cout << *it << " ";

    return 0;
}
```

Output

```
0 1 1
1 2 2
2 3 4
3 4 4
4 5 8
5 6 8
6 7 8
7 8 8
8 9 16
9 10 16
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

STL Sequence Containers

- Sequence containers
  - list (https://www.geeksforgeeks.org/list-cpp-stl/)
    - #include<list>
    - Allocated memory when element is added (does not reserve any extra memory like vector)
    - Extra memory is needed to store links
    - Does not allows for random access of elements. But allows bidirectional access
    - Adding/removing element from front or back or middle will take constant time (when you know position where to add or remove)

```
int main()
{
    std::list<float> float_list;

    for(int i = 0; i < 5; i++) {
        float_list.push_back(i+1);
        float_list.push_front((i+1) *0.1);
        cout << i << " ";
        cout << float_list.size() << endl;
    }

    // for(int i = 0; i < 10; i++) {
    //     error: no match for ‘operator[]’
    //     cout << float_list[i] << " ";
    // }
    // cout << endl;
    // Ignore following for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = float_list.cbegin(); it != float_list.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = float_list.crbegin(); it != float_list.crend(); ++it)
        cout << *it << " ";

    return 0;
}
```

Output

```
0 2
1 4
2 6
3 8
4 10
0.5 0.4 0.3 0.2 0.1 1 2 3 4 5
5 4 3 2 1 0.1 0.2 0.3 0.4 0.5
```

STL Sequence Containers

- Sequence containers
  - forward_list (https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)
    - #include<forward_list>
    - Use only when you need to store very few elements
      - Designed with that in mind (e.g. does not give size)
    - Does not allow random or bidirectional access. Only allows forward access
    - Adding/removing element from front or back or middle will take constant time (when you know position where to add or remove)
      - It does not have push_back method because unlike list it does not remember location of the last node.

```
int main()
{
    std::forward_list<float> float_flist;
    for(int i = 0; i < 10; i++) {
        // error: forward_list has no member named ‘push_back’
        // float_flist.push_back(i+1);
        float_flist.push_front((i+1) *0.1);
        // error: forward_list has no member named ‘size’
        // cout << float_flist.size() << endl;
    }
    // for(int i = 0; i < 10; i++) {
    //     error: no match for ‘operator[]’
    //     cout << float_flist[i] << " ";
    // }
    // cout << endl;
    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = float_flist.cbegin(); it != float_flist.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    // error: forward_list has no member named 'crbegin' and 'crend'
    // for(auto it = float_flist.crbegin(); it != float_flist.crend(); ++it)
    //    cout << *it << " ";
    return 0;
}
```

Output

```
1 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1
```

STL Sequence Containers

- Sequence containers
  - deque (https://www.geeksforgeeks.org/deque-cpp-stl/)
    - #include<deque>
    - Allows random access of elements in constant time using [ ]
    - Allows insertion and removal of elements at two two ends
      - Its done in constant time as it does not need reallocation
      - https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl
    - Does not guarantee contiguous memory allocation for storing elements
    - Adding/removing elements from the middle is costly (O(n))

```
int main()
{
    std::deque<int> int_deque;

    for(int i = 0; i < 5; i++) {
        int_deque.push_back(i+1);
        int_deque.push_front((i+1) * 2);
        cout << i << " ";
        cout << int_deque.size() << endl;
    }

    for(int i = 0; i < 10; i++) {
        cout << int_deque[i] << " ";
    }
    cout << endl;
    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = int_deque.cbegin(); it != int_deque.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = int_deque.crbegin(); it != int_deque.crend(); ++it)
        cout << *it << " ";

    return 0;
}
```

Output

```
0 2
1 4
2 6
3 8
4 10
10 8 6 4 2 1 2 3 4 5
10 8 6 4 2 1 2 3 4 5
5 4 3 2 1 2 4 6 8 10
```

STL Associative Containers

- Associative containers
  - Ordered associative containers
    - map - An ordered map (key-value pairs)
    - multimap - An ordered map (key-value pairs). Duplicate keys allowed
    - set - An ordered set
    - multiset - An ordered set. Duplicates allowed
    - **Note**: usually implemented as balanced binary trees (usually red-black trees)
  - Unordered associative containers
    - unordered_map - An unordered map (key-value pairs)
    - unordered_multimap - An unordered map (key-value pairs). Duplicate keys allowed
    - unordered_set - An unordered set
    - unordered_multiset - An unordered set. Duplicates allowed
    - **Note**: usually implemented as hash tables with linked overflow

```
int main()
{
    std::set<int> set1;
    for(int i = 0; i < 10; i++) {
        int random_number = rand() % 100;
        cout << random_number << " ";
        // insert method will not insert duplicates
        set1.insert(random_number);
    }
    cout << endl;
    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = set1.cbegin(); it != set1.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = set1.crbegin(); it != set1.crend(); ++it)
        cout << *it << " ";

    cout << endl << set1.count(86);

    return 0;
}
```

Output

```
83 86 77 15 93 35 86 92 49 21
15 21 35 49 77 83 86 92 93
93 92 86 83 77 49 35 21 15
1
```

- Need to include ***set***
- Note that there are only 9 numbers in the set
    - Bacause 86 is being generated twice by rand() functon and insert method does not insert it twice in the set

```
int main()
{
    std::multiset<int> set1;
    for(int i = 0; i < 10; i++) {
        int random_number = rand() % 100;
        cout << random_number << " ";

        set1.insert(random_number);
    }
    cout << endl;
    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = set1.cbegin(); it != set1.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = set1.crbegin(); it != set1.crend(); ++it)
        cout << *it << " ";

    cout << endl << set1.count(86);

    return 0;
}
```

Output

```
83 86 77 15 93 35 86 92 49 21
15 21 35 49 77 83 86 86 92 93
93 92 86 86 83 77 49 35 21 15
2
```

- Need to include ***set***
- Note that there are 10 numbers in the multiset
    - Bacause 86 is being generated twice by rand() functon and insert method inserts it twice in the multiset

```
int main()
{
    std::map<int, string> map1;
    map1.insert(std::pair<int, string>(10, "Nadiad"));
    map1.insert(std::pair<int, string>(32, "Kapadwanj"));
    map1.insert(std::pair<int, string>(7, "Dakor"));
    map1.insert(std::pair<int, string>(7, "Ahmedabad"));
    map1.insert(std::pair<int, string>(23, "Vadodara"));

    cout << map1[7] << endl << endl;

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = map1.cbegin(); it != map1.cend(); ++it)
        cout << (*it).first << " " << it->second << endl;
    cout << endl;
    for(auto it = map1.crbegin(); it != map1.crend(); ++it)
        cout << (*it).first << " " << it->second << endl;

   cout << endl << map1.count(7);

    return 0;
}
```

Output

```
Dakor

7 Dakor
10 Nadiad
23 Vadodara
32 Kapadwanj

32 Kapadwanj
23 Vadodara
10 Nadiad
7 Dakor

1
```

- Need to include ***map***
- std::map does not allow duplicate keys
- std::map overloads [ ] operator. It returns value associated with given key

```
int main()
{
    std::multimap<int, string> map1;
    map1.insert(std::pair<int, string>(10, "Nadiad"));
    map1.insert(std::pair<int, string>(32, "Kapadwanj"));
    map1.insert(std::pair<int, string>(7, "Dakor"));
    map1.insert(std::pair<int, string>(7, "Ahmedabad"));
    map1.insert(std::pair<int, string>(23, "Vadodara"));

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = map1.cbegin(); it != map1.cend(); ++it)
        cout << (*it).first << " " << it->second << endl;
    cout << endl;
    for(auto it = map1.crbegin(); it != map1.crend(); ++it)
        cout << (*it).first << " " << it->second << endl;

   cout << endl << map1.count(7);

    return 0;
}
```

Output

```
7 Dakor
7 Ahmedabad
10 Nadiad
23 Vadodara
32 Kapadwanj

32 Kapadwanj
23 Vadodara
10 Nadiad
7 Ahmedabad
7 Dakor

2
```

- Need to include ***map***
- std::multimap allows duplicate keys
  - If there are duplicate keys, multimap will preserve insert order of key-value pair (since C++11)
- No overload of [ ]

```
int main()
{
    std::unordered_set<int> set1;

    for(int i = 0; i < 10; i++) {
        int random_number = rand() % 100;
        cout << random_number << " ";
        // insert method will not insert duplicates
        set1.insert(random_number);
    }
    cout << endl;

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = set1.cbegin(); it != set1.cend(); ++it)
        cout << *it << " ";

    cout << endl << set1.count(86);

    return 0;
}
```

Output

```
83 86 77 15 93 35 86 92 49 21
21 92 93 77 86 35 49 83 15
1
```

- Need to include ***unordered_set***
- std::unordered_set allows only forward access
- Does not allow duplicate entries
- Elements are not stored in any particular order

```
int main()
{
    std::unordered_multiset<int> set1;

    for(int i = 0; i < 10; i++) {
        int random_number = rand() % 100;
        cout << random_number << " ";

        set1.insert(random_number);
    }
    cout << endl;

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = set1.cbegin(); it != set1.cend(); ++it)
        cout << *it << " ";

    cout << endl << set1.count(86);

    return 0;
}
```

Output

```
83 86 77 15 93 35 86 92 49 21
21 92 93 77 86 86 35 49 83 15
2
```

- Need to include ***unordered_set***
- std::unordered_multiset allows only forward access
- Allows duplicate entries
- Elements are not stored in any particular order

```
int main()
{
   std::unordered_map<int, string> map1;

   map1.insert(std::pair<int, string>(10, "Nadiad"));
   map1.insert(std::pair<int, string>(32, "Kapadwanj"));
   map1.insert(std::pair<int, string>(7, "Dakor"));
   map1.insert(std::pair<int, string>(7, "Ahmedabad"));
   map1.insert(std::pair<int, string>(23, "Vadodara"));

   // Ignore this for loop for now
   // You will understand this code once we discuss iterators
   for(auto it = map1.cbegin(); it != map1.cend(); ++it)
      cout << (*it).first << " " << it->second << endl;

   cout << endl << map1.count(7);

   return 0;
}
```

Output

```
23 Vadodara
7 Dakor
10 Nadiad
32 Kapadwanj

1
```

- Need to include ***unordered_map***
- std::unordered_map allows only forward access
- It does not allow duplicate keys
- Elements are not stored in any particular order
- std::unordered_map overloads [ ] operator. It returns value associated with given key

```
int main()
{
    std::unordered_multimap<int, string> map1;

    map1.insert(std::pair<int, string>(10, "Nadiad"));
    map1.insert(std::pair<int, string>(32, "Kapadwanj"));
    map1.insert(std::pair<int, string>(7, "Dakor"));
    map1.insert(std::pair<int, string>(7, "Ahmedabad"));
    map1.insert(std::pair<int, string>(23, "Vadodara"));

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(auto it = map1.cbegin(); it != map1.cend(); ++it)
        cout << (*it).first << " " << it->second << endl;

    cout << endl << map1.count(7);

    return 0;
}
```

Output

```
23 Vadodara
7 Ahmedabad
7 Dakor
10 Nadiad
32 Kapadwanj

2
```

- Need to include ***unordered_map***
- std::unordered_multimap allows only forward access
- It allows duplicate keys
- Elements are not stored in any particular order

Container adaptors

- A container adaptor provides a different (typically restricted) interface to a container.
- They do not offer iterators or subscripting.
- Stack
  - The stack container adaptor is defined in <stack>
  - stack is an interface to a container of the type passed to it as a template argument
  - A stack eliminates the non-stack operations on its container from the interface, and provides the conventional names: top() , push() , and pop()
  - By default, a stack makes a deque to hold its elements, but any sequence that provides back(), push_back() , and pop_back() can be used. For example:
    - stack<char> s1; // uses a deque<char> to store elements
    - stack<int,vector<int>> s2; // uses a vector<int> to store elements

```
int main()
{
    std::stack<string> stack1;

    stack1.push(string("ABC"));
    stack1.push(string("PQR"));
    stack1.push(string("XYZ"));
    cout << stack1.top() << endl;
    stack1.pop();
    cout << stack1.top() << endl;
    stack1.pop();

    return 0;
}
```

Output

```
XYZ
PQR
```

- Need to include ***stack***
- std::stack is a container adaptor and does not provide any iterator
- Only LIFO access is allowed

```
int main()
{
   std::queue<string> stack1;

   stack1.push(string("ABC"));
   stack1.push(string("PQR"));
   stack1.push(string("XYZ"));
   cout << stack1.front() << endl;
   stack1.pop();
   cout << stack1.front() << endl;
   stack1.pop();

   return 0;
}
```

Output

```
ABC
PQR
```

- Need to include ***queue***
- std::queue is a container adaptor and does not provide any iterator
- Only FIFO access is allowed

```
int main()
{
   std::priority_queue<int> stack1;

   stack1.push(10);
   stack1.push(20);
   stack1.push(5);
   stack1.push(15);
   cout << stack1.top() << endl;
   stack1.pop();
   cout << stack1.top() << endl;
   stack1.pop();
   cout << stack1.top() << endl;

   return 0;
}
```

Output

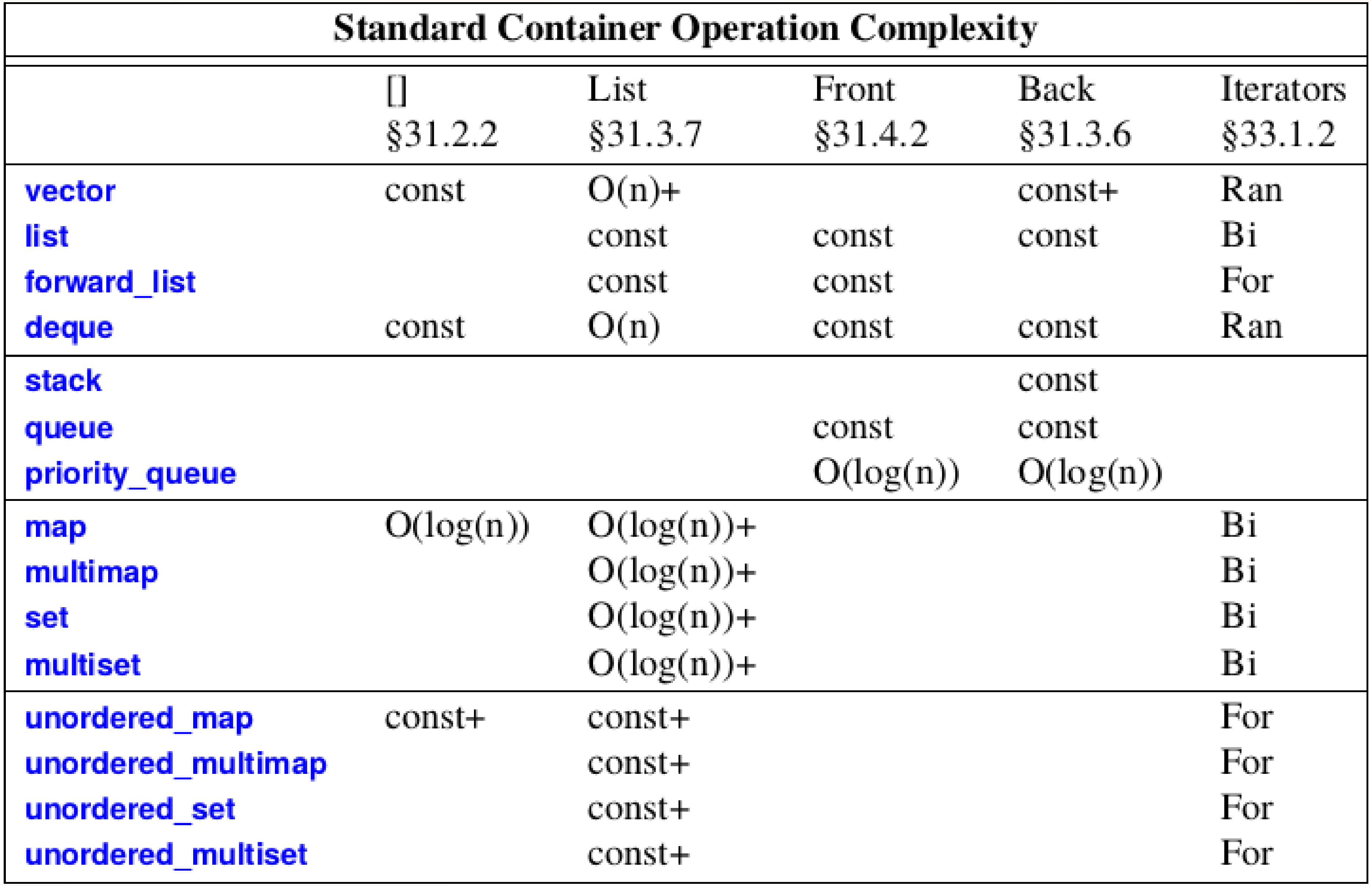
```
20
15
10
```

- Need to include ***queue***
- std::priority_queue is a container adaptor and does not provide any iterator

```
class Person {
public:
    int age;
    float salary;
    Person(int age, float salary) {
        this->age = age;
        this->salary = salary;
    }
};
bool operator<(const Person p1, const Person &p) {
    return p1.salary < p.salary;
}
int main()
{
    std::priority_queue<Person> pri_que;
    pri_que.push(Person(32, 300000));
    pri_que.push(Person(22, 600000));
    pri_que.push(Person(32, 300000));
    pri_que.push(Person(42, 500000));
    pri_que.push(Person(22, 800000));
    while (!pri_que.empty()) {
        Person p = pri_que.top();
        pri_que.pop();
        cout << p.age << " " << p.salary << "\n";
    }
    return 0;
}
```

Output

```
22 800000
22 600000
42 500000
32 300000
32 300000
```



Standard Container Operation Complexity

	[]	List	Front	Back	Iterators
	§31.2.2	§31.3.7	§31.4.2	§31.3.6	§33.1.2
vector	const	O(n)+		const+	Ran
list		const	const	const	Bi
forward_list		const	const		For
deque	const	O(n)	const	const	Ran
stack				const	
queue			const	const	
priority_queue			O(log(n))	O(log(n))	
map	O(log(n))	O(log(n))+			Bi
multimap		O(log(n))+			Bi
set		O(log(n))+			Bi
multiset		O(log(n))+			Bi
unordered_map	const+	const+			For
unordered_multimap		const+			For
unordered_set		const+			For
unordered_multiset		const+			For

Front – insertion/deletion before first element

Back – insertion/deletion after last element

List – insertion/deletion not necessarily at the ends

Ran – Random access iterator

For – Forward iterator

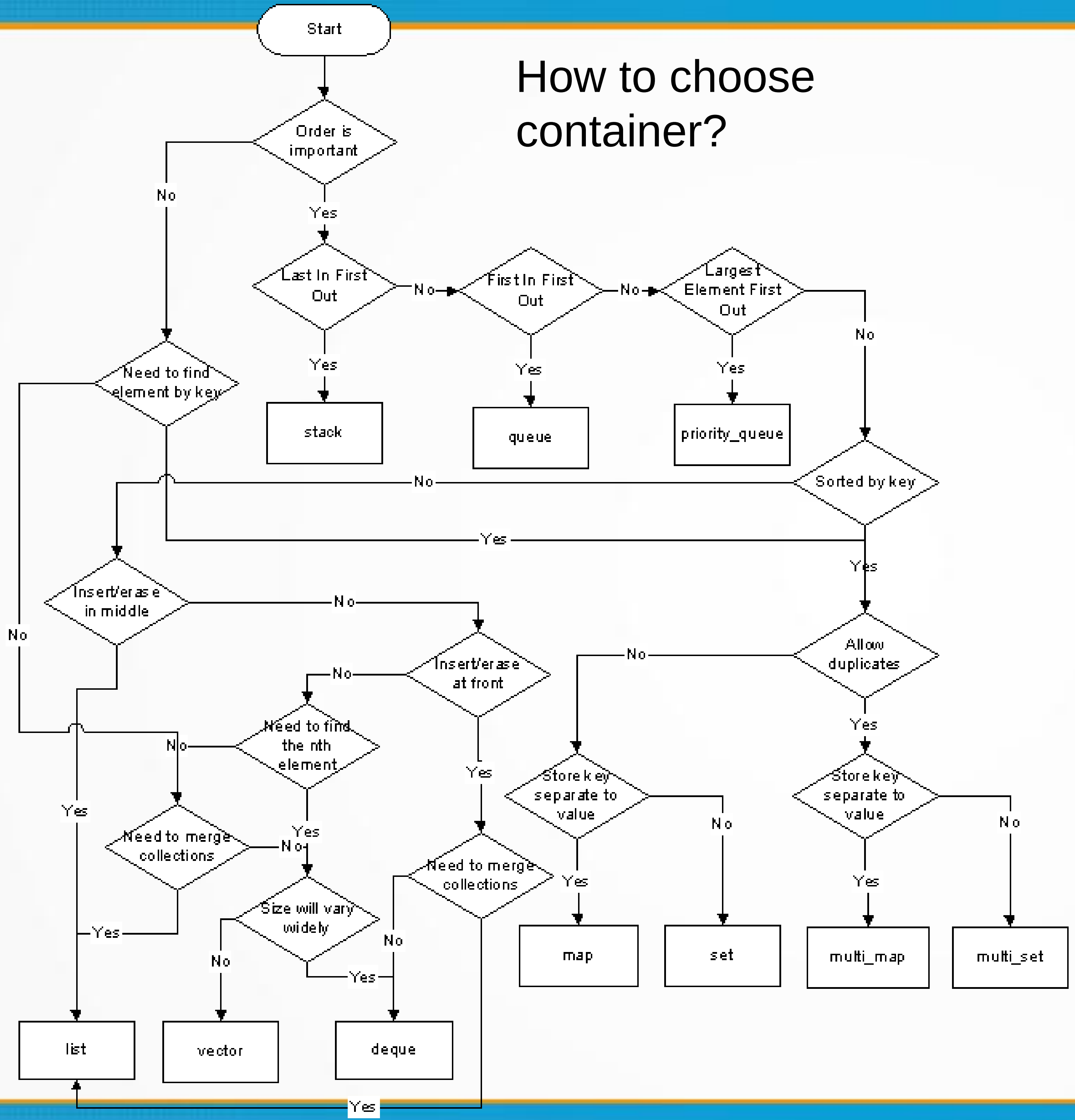
Bi – Bidirectional iterator

const – time taken is independent of number of elements in the list

Const < O(log(n)) < O(n)

Source: “The C++ programming language”, fourth edition, by Bjarne Stroustrup

How to choose container?



Source:
https://web.archive.org/web/20180824133558/homepages.e3.net.nz/~djm/cppcontainers.html

Created by David Moore
Licensed CC BY-SA 3.0

Iterators

- **Forward iterators:**
  - We can iterate forward repeatedly using ++
  - Allows read and write using *
  - If a forward iterator points to a class object, we can use −> to refer to a member
  - We can compare forward iterators using == and !=
- **Bidirectional iterator:**
  - It has all the capabilities of forward iterator
  - We can iterate forward (using ++) and backward (using −−)
- **Random-access iterator:**
  - It has all the capabilities of bidirectional iterator
  - We can add an integer using + , and subtract an integer using −
  - We can iterate forward (using ++ or +=) and backward (using − or −=)
  - Allows read and write using ∗ or [ ]
  - We can find the distance between two random-access iterators to the same sequence by subtracting one from the other
  - We can compare random-access iterators using == , != , < , <= , > , and >=

```
int main()
{
    std::vector<int> int_vect;

    for(int i = 0; i < 10; i++)
        int_vect.push_back(i+1);

   for(int i = 0; i < 10; i++)
        cout << int_vect[i] << " ";
    cout << endl;

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(std::vector<int>::iterator it = int_vect.begin(); it != int_vect.end(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = int_vect.rbegin(); it != int_vect.rend(); ++it)
        cout << *it << " ";

    return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

```
int main()
{
    std::vector<int> int_vect;

    for(int i = 0; i < 10; i++)
        int_vect.push_back(i+1);

   for(int i = 0; i < 10; i++)
        cout << int_vect[i] << " ";
    cout << endl;

    // Ignore this for loop for now
    // You will understand this code once we discuss iterators
    for(std::vector<int>::const_iterator it = int_vect.cbegin(); it != int_vect.cend(); ++it)
        cout << *it << " ";
    cout << endl;
    for(auto it = int_vect.crbegin(); it != int_vect.crend(); ++it)
        cout << *it << " ";

    return 0;
}
```

Output

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

```
int main()
{
    std::vector<int> int_vect;

    for(int i = 0; i < 10; i++)
        int_vect.push_back(i+1);

   auto itr = std::find(int_vect.begin(), int_vect.end(), 4);
   cout << *itr;

    return 0;
}
```

Output

```
4
```

- Need to include ***algorithm***

```
int main()
{
    std::list<float> float_list;

    for(int i = 0; i < 5; i++) {
        float_list.push_back(i+1);
        float_list.push_front((i+1) *0.1);
    }

   auto itr = std::find(float_list.begin(), float_list.end(), 4);
   cout << *itr;

    return 0;
}
```

Output

```
4
```

- Need to include ***algorithm***

```
int main()
{
    std::vector<int> int_vect;

    for(int i = 10; i > 0; i--)
        int_vect.push_back(i);

    for(auto item: int_vect)
        cout << item << " ";
    cout << endl;

    std::sort(int_vect.begin(), int_vect.end());

    for(auto item: int_vect) {
        cout << item << " ";
    }

    return 0;
}
```

Output

```
10 9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 10
```

- Need to include ***algorithm***

```
int main()
{
    std::deque<int> int_deque;

    for(int i = 0; i < 5; i++) {
        int_deque.push_back(i+1);
        int_deque.push_front((i+1) * 2);
    }

    for(auto item: int_deque)
        cout << item << " ";
    cout << endl;

    std::sort(int_deque.begin(), int_deque.end());

    for(auto item: int_deque) {
        cout << item << " ";
    }

    return 0;
}
```

Output

```
10 8 6 4 2 1 2 3 4 5
1 2 2 3 4 4 5 6 8 10
```

- Need to include ***algorithm***

Interesting reads

- Bound check is guaranteed for vector if elements are access using at(), not guaranteed when elements are access with [ ]
  - https://stackoverflow.com/questions/16620222/vector-going-out-of-bounds-without-giving-error
- Type members
  - https://stackoverflow.com/questions/14037392/how-are-member-types-implemented
- How is deque implemented?
  - https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl
- Iterators
  - https://users.cs.northwestern.edu/~riesbeck/programming/c++/stl-iterators.html