Lecture 11.04

Standard Template Library

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides from Prof. Shin at DGIST

Short Notice

- Uploaded the midterm score with the solution
  - Claim: Today, 4:00~6:00pm, E3 613
- Will upload HW2 score and sample solutions written by students today
  - I will review the sample solutions next week if we have free time
- Will release HW4 on next Monday
  - I will review the HW4 on next Monday during the lecture time
  - The due will be 12/02 Wednesday, around one month later!
  - But I promise – you will be also busy because of the team project, so be prepared earlier

Today's Topic

- Something useful for your projects (cont. from the last lecture)
  - String stream
  - Smart Pointer
- Containers
  - Vector
  - Map
- Iterators
  - iterator
  - reverse_iterator

std::stringstream

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <iomanip>

int main() {
    std::stringstream os;
    os << std::setw(10) << std::left << "Alice" << '|';
    os << std::setw(10) << std::right << 27 << '|';
    os << std::setw(12) << "053-785-6684" << '|';

    std::string data = os.str();
    std::cout << data;
}
```

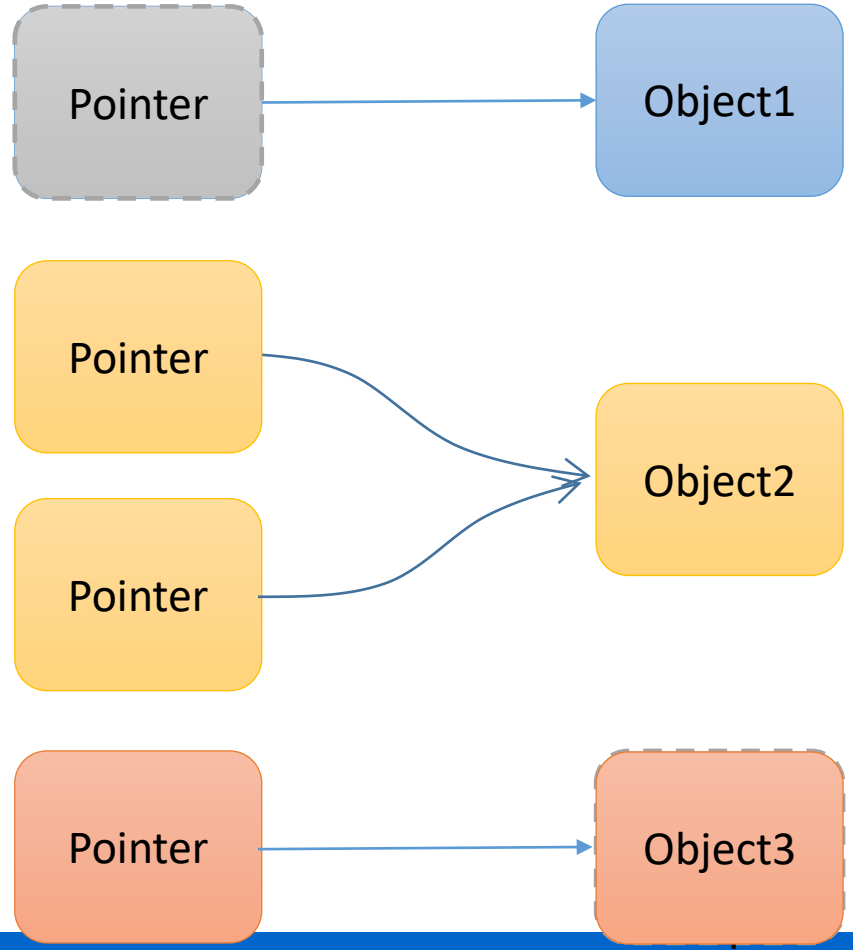
```
Alice     |        27|053-785-6684|
```

Smart Pointer

- Problems on using pointers
  - Memory leak
  - Dangling pointer

- Smart pointer
  - auto_ptr (deleted since c++11)
  - std::unique_ptr (c++11)
  - std::shared_ptr (c++11)
  - std::weak_ptr (c++11)

Please Google It!



Example: Smart Pointer

```cpp
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> p1(new int(10));
    std::unique_ptr<int> p2 = p1; // error
    std::unique_ptr<int> p3 = std::move(p1);
    auto& a = *p3;
    std::cout << a << std::endl;

    p3.reset();
    p1.reset();
}
```

```cpp
#include <memory>
int main() {
    std::shared_ptr<int> p1(new int(10));
    std::shared_ptr<int> p2 = p1;

    auto& a = *p2;
    cout <<“value: "<<a<<" owner: "<<p1.use_count();
    p2.reset();
    cout <<"\nvalue: "<<a<<" owner: "<<p1.use_count();
    p1.reset();
    cout <<"\nvalue: "<<a<<" owner: "<<p1.use_count();
}
```

STL: Standard Template Library

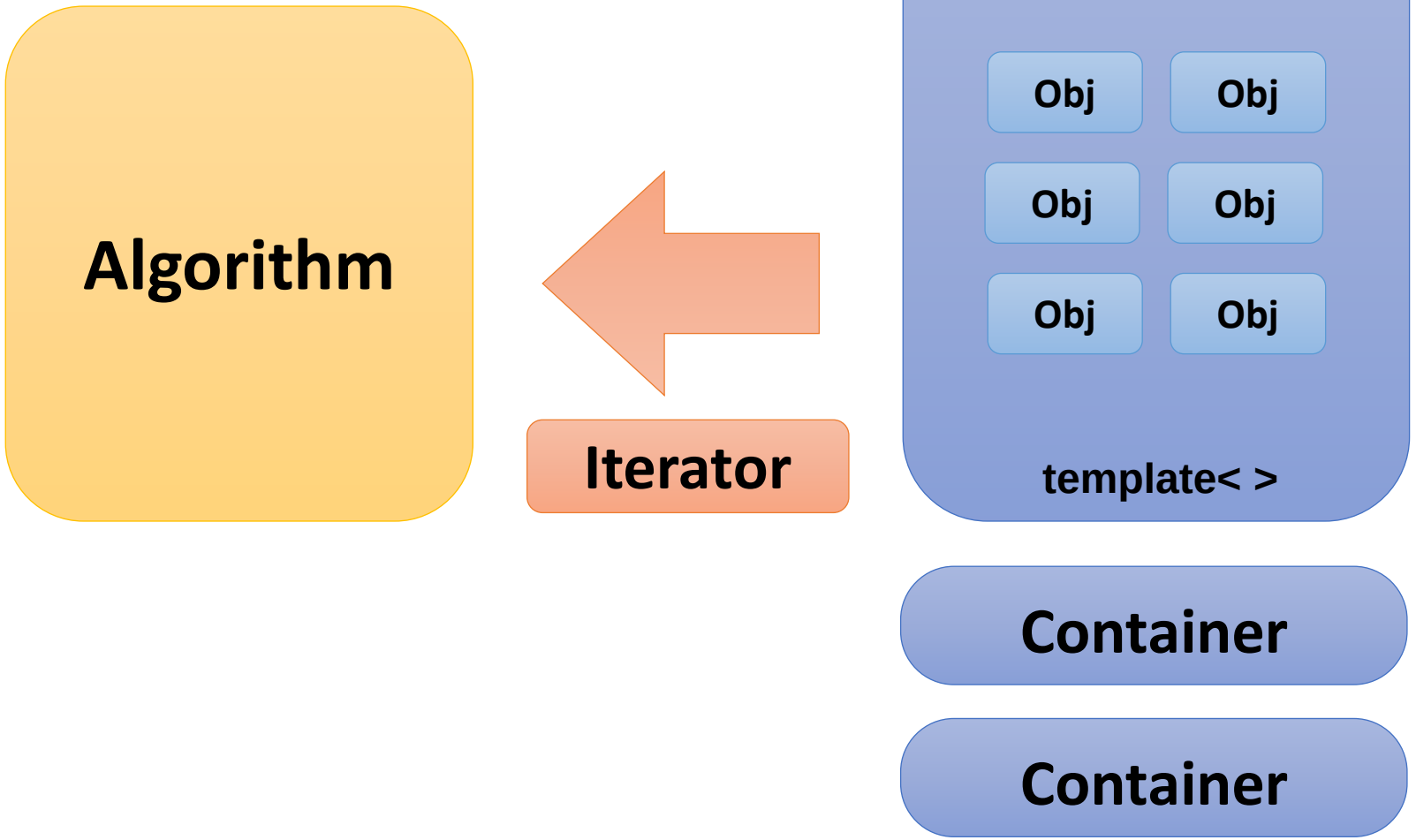
- The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library
- It provides four components called **algorithms, containers, functions, and iterators**[1]

- History
  - STL was introduced by Alex Sepanov and Meng Lee (HP lab) in 1994
  - Later, STL was included in C++ standard template library by ISO/ANSI C++ standard committee

[1] wikipedia

STL: Standard Template Library

- Main components
  - Container
  - Iterator
  - Algorithm
  - Function object
  - Adaptor
  - Allocator



Containers

- Containers manage **collection of the elements** (of the same data type)
  - Containers share similar (but not exactly the same) interfaces
  - Each container has different data structure to store elements, different memory and time overhead

  →You need to select a proper container for your purpose
- Containers provide similar interfaces (i.e., public member functions) for **generic programming** as well as easy maintenance

Types of Containers

- Sequential containers
  - vector, list, forward_list, deque, array

- Associative containers
  - set, multiset, map, multimap
- Unordered containers
  - unordered_set, unordered_multiset, unordered_map, unordered_multimap

Sequential Containers

- vector

Array-based containers

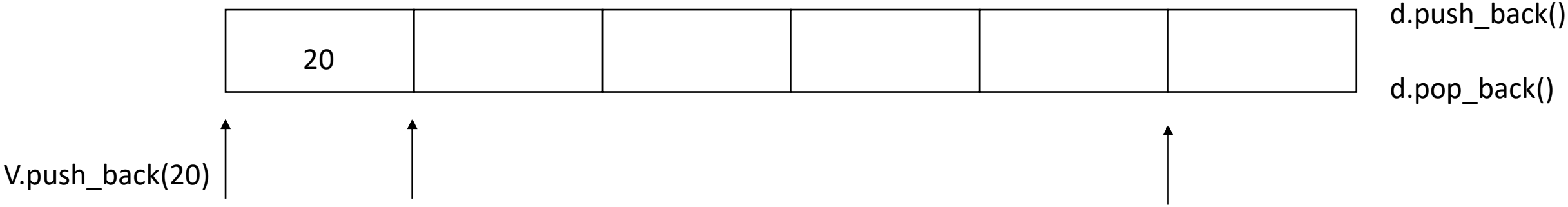
- deque

- list

Node-based containers

data is sequentially stored according to input sequence

Vector



Sequential Containers

- vector

Array-based containers

- deque

- list

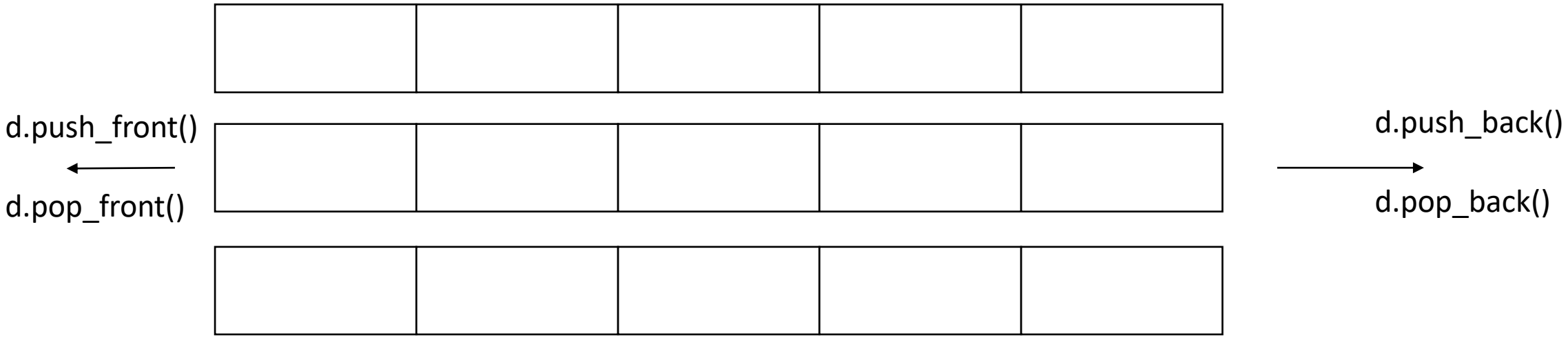
Node-based containers

data is sequentially stored according to input sequence

Q1. memory reallocation?

Q2. push_front ?

double-**e**nded **que**ue



Sequential Containers

- vector

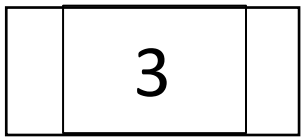
Array-based containers

- deque

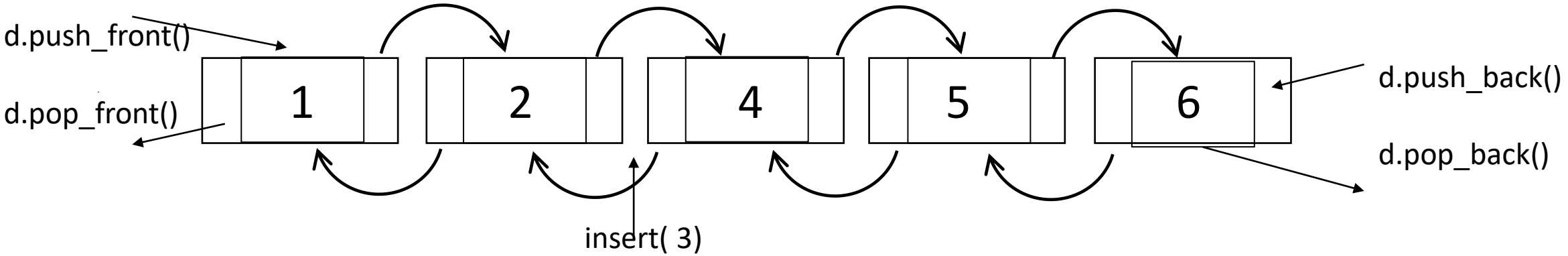
- list

Node-based containers

data is sequentially stored according to input sequence



Doubly-linked list



Vector

- Constructors
  - vector v : empty vector
  - vector v(n): n-sized vector
  - vector v(n,x): n-sized vector with element x
  - vector v(v2): copy constructor
  - vector v(b,e): initialized with **iterator** range[b, e)

```
#include <iostream>
#include <vector>
using namespace std;
class Shape;
int main() {
    vector<int> v1{ 1, 2, 3, 4 };
    vector<string> v2;
    vector<Shape*> v3(23);
    vector<double> v4(32,9.9);
}
```

Vector

- Member methods
  - at(i):
  - back()
  - front()
  - capacity()
  - size()
  - empty()
  - shrink_to_fit()
  - push_back(i)
  - pop_back()
  - operator[]
  - clear()
  - insert(p, x)
  - begin()
  - end()

https://en.cppreference.com/w/cpp/container/vector

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

Example: Vector

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v1;
    v1.push_back(1);
    v1.push_back(2);
    std::vector<int> v2(v1);
    v1[0] = 3;
    v1.pop_back();
    for (int& a : v1) {
        std::cout << a << " ";
    }
}
```

```
3
```

Iterators

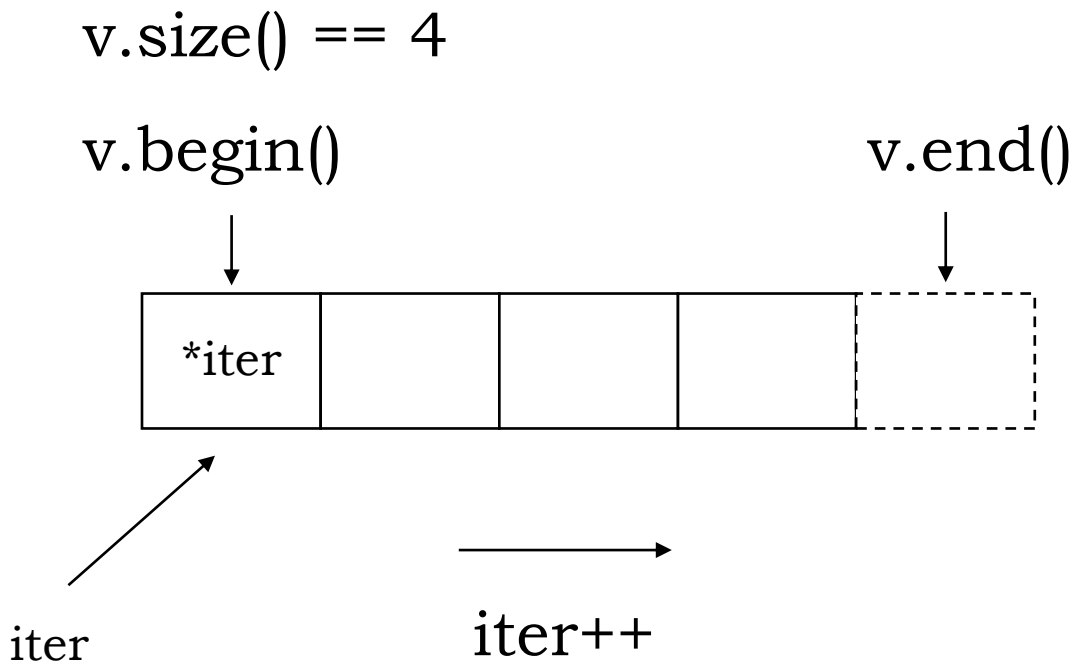
- A **pointer-like** object with the following operators
  - T(a): copy iterator (copy constructor)
  - ++, --: increment/decrement to indicate the next/previous object in a given container
  - *, ->: dereference (i.e., the object indicated by the iterator)
  - ==, !=: test for equality or inequality
  - c.f., no assignment operator
- Iterator provides common/similar interface to access various containers

Iterator in the Container

- x::iterator, x::const_iterator
- [begin, end)

```
vector<int>::iterator iter;
vector<int>::const_iterator citer;

iter = v.begin();
cout << *iter;
```



Example: iterator

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v{ 1,2,3,4 };
    vector<int>::iterator iter;
    for (iter = v.begin(); iter != v.end(); ++iter) {
        cout << *iter << " ";
        *iter -= 1;
    }
    vector<int>::const_iterator citer{ iter };
    // cout << *iter;

    for (citer = v.begin(); citer != v.end(); ++citer) {
        cout << *citer << " ";
        //*citer -= 1;
    }
}
```

Example: range-based for with iterator and auto

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> v{ 1,2,3,4 };
// for (int i=0; i < v.size(); ++i) {       v[i] ...  }
    for (vector<int>::iterator iter = v.begin(); iter != v.end(); ++iter) {
        cout << *iter << " ";
    }
    for (auto iter = v.begin(); iter != v.end(); ++iter) {
        cout << *iter << " ";
    }
    for (auto & e : v) cout << e << " ";
}
```

References

- Learn C++ (https://www.learncpp.com/)
  - STL: Ch. 16

- STL
  - http://en.cppreference.com/w/cpp/container
  - http://en.cppreference.com/w/cpp/iterator

ANY QUESTIONS?

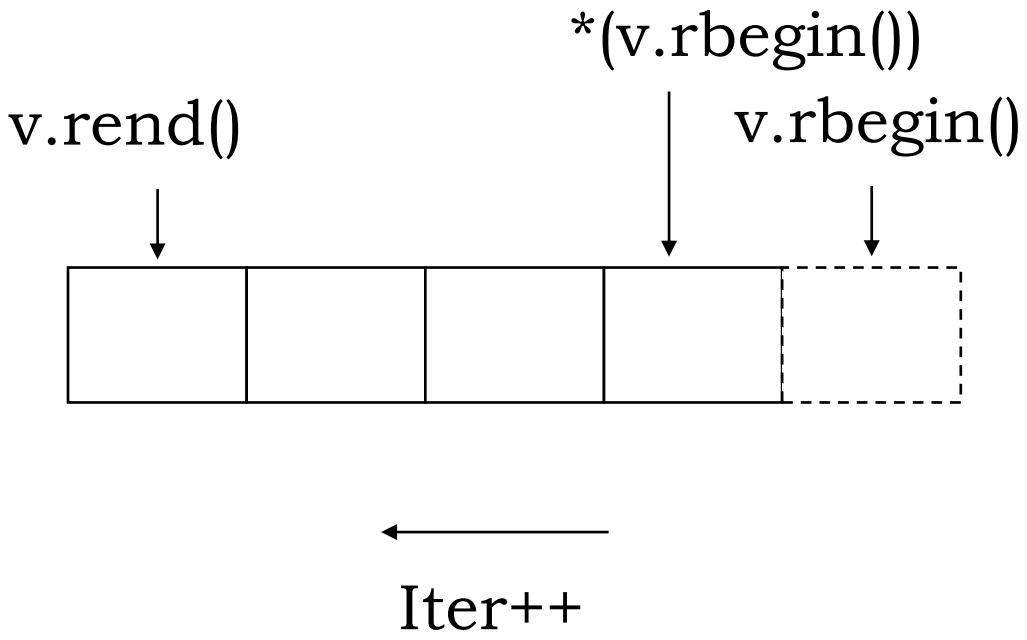
Backup slides exist. Study by yourself!

Reverse Iterator in the Container

- x::reverse_iterator, x::const_reverse_iterator
- [rbegin, rend)

```
vector<int>::iterator iter;
vector<int>::const_iterator citer;


iter = v.begin();
cout << *iter;
```



Example: reverse_iterator

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v{ 1,2,3,4 };
    vector<int>::iterator iter = v.begin()+1;
    vector<int>::const_iterator citer{ iter };
    // vector<int>::reverse_iterator riter(iter);
    reverse_iterator<vector<int>::iterator> riter(iter);
    cout << *iter << " " << *citer << " " << *riter;
}
```