

STL

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

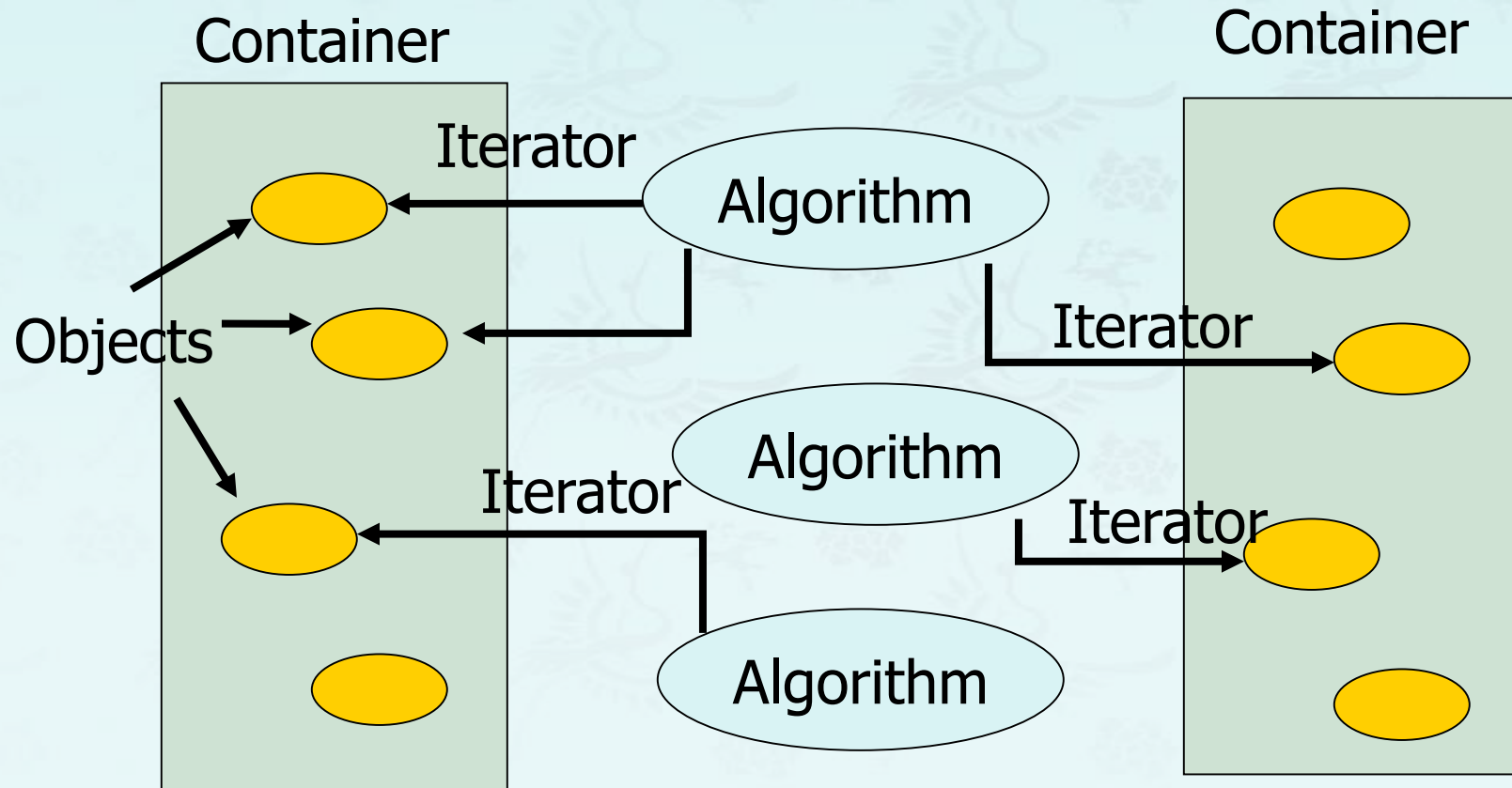
Standard Template Library

Standard Template Library

- The standard template library (STL) contains
 - Containers
 - Algorithms
 - Iterators
- **Containers** are **generic** class templates for storing collection of data, for example an array of elements.
- **Algorithms** are **generic** function templates for operating on containers, for example search for an element in an array, or sort an array.
- **Iterators** are generalized '**smart**' **pointers** that facilitate use of containers, for example you can increment an iterator to point to the next element in an array.

Containers, Iterators, Algorithms

- Algorithms use iterators to interact with objects stored in containers



Containers

- A container is a way to store data, either built-in data types like int and float, or class objects
- The STL provides several basic kinds of containers
 - **<vector>** : one-dimensional array
 - **<list>** : double linked list
 - **<deque>** : double-ended queue
 - **<queue>** : queue
 - **<stack>** : stack
 - **<set>** : set
 - **<map>** : associative array

Containers

STL 컨테이너	특 징
vector	<ul style="list-style-type: none"> - 동적 배열이므로 배열의 크기를 변경할 수 있다. - 임의 접근이 가능하며, 뒤에서의 삽입이 빠르다.
list	<ul style="list-style-type: none"> - 연결 리스트이므로 데이터를 순차적으로 접근하고 관리할 때 유용하다. - 위치에 상관없이 삽입과 삭제가 빠르다.
deque	<ul style="list-style-type: none"> - 데크라고 한다. - 임의 접근이 가능하며, 앞과 뒤에서의 삽입이 빠르다.
map	<ul style="list-style-type: none"> - 특정 키(key)에 의해서 데이터를 접근하고 관리할 수 있다 - 키를 통해 값을 접근하며, 삽입과 삭제가 빠르다.
set	<ul style="list-style-type: none"> - 원소들을 순서대로 관리하며, 소속 검사와 삽입, 삭제가 빠르다. - 중복된 원소를 허용하지 않는다.
stack	<ul style="list-style-type: none"> - top에서만 삽입과 삭제가 가능하다. - LIFO(Last In First Out) 방식으로 데이터를 삽입, 삭제 한다.
queue	<ul style="list-style-type: none"> - 삽입은 뒤쪽에서, 삭제는 앞쪽에서 수행한다. - FIFO(First In First Out) 방식으로 데이터를 삽입, 삭제 한다.

Sequence 순차 Containers

Associative 연관 Containers

Adaptor Containers

Sequence Containers

- A **sequence container** stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another, **<vector>**, **<list>** and **<deque>** are sequential containers.

Sequence Containers

- A **sequence container** stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another, **<vector>**, **<list>** and **<deque>** are sequential containers.
- In an ordinary C++ array the size is fixed and can not change during run-time, it is also tedious to insert or delete elements.
Advantage: quick random access

Sequence Containers

- A **sequence container** stores a set of elements in sequence, in other words each element (except for the first and last one) is preceded by one specific element and followed by another, **<vector>**, **<list>** and **<deque>** are sequential containers.
- In an ordinary C++ array the size is fixed and can not change during run-time, it is also tedious to insert or delete elements.
Advantage: quick random access
- **<vector> is an expandable array** that can shrink or grow in size, but still has the **disadvantage** of inserting or deleting elements in the middle

Sequence Containers

- **<list>** is a **double linked list** (each element has points to its successor and predecessor), it is quick to insert or delete elements but has slow random access
- **<deque>** is a **double-ended queue**, that means one can insert and delete elements from both ends.
It is a kind of combination between a stack (last in first out) and a queue (first in first out) and constitutes a compromise between a **<vector>** and a **<list>**

Associative Containers

- **An associative container** is non-sequential but uses a **key** to access elements. The keys, typically a number or a string, are used by the container to arrange the stored elements in a specific order. For example in a dictionary the entries are ordered alphabetically.

Associative Containers

- A **<set>** stores a number of items which contain keys. The keys are the attributes used to order the items. For example, a set might store objects of the class Person which are ordered alphabetically using their name.
- A **<map>** stores pairs of objects: a key object and an associated value object. A <map> is somehow similar to an array except instead of accessing its elements with index numbers, you access them **with indices of an arbitrary type**.
- <set> and <map> only allow one key of each value, whereas **<multiset>** and **<multimap>** allow multiple identical key values.

Vector Container

- Provides an alternative to the built-in array.
- A vector is self grown.
- Use it instead of the built-in array!
- For example:
 - vector<int> - vector of integers.
 - vector<string> - vector of strings.
 - vector<int * > - vector of pointers to integers.
 - vector<**Shape**> - vector of Shape objects. **Shape is a user defined class.**

Operations on vector

- iterator **begin**();
- iterator **end**();
- bool **empty**();
- void **push_back**(const T& x);
- iterator **erase**(iterator it);
- iterator **erase**(iterator first, iterator last);
- void **clear**();
-

Vector Container Example

```
#include<iostream>
#include<vector>
using namespace std;
int main(){
    vector<int> v(5);
    for(int i=0; i < v.size(); i++)
        cin >> v[i];

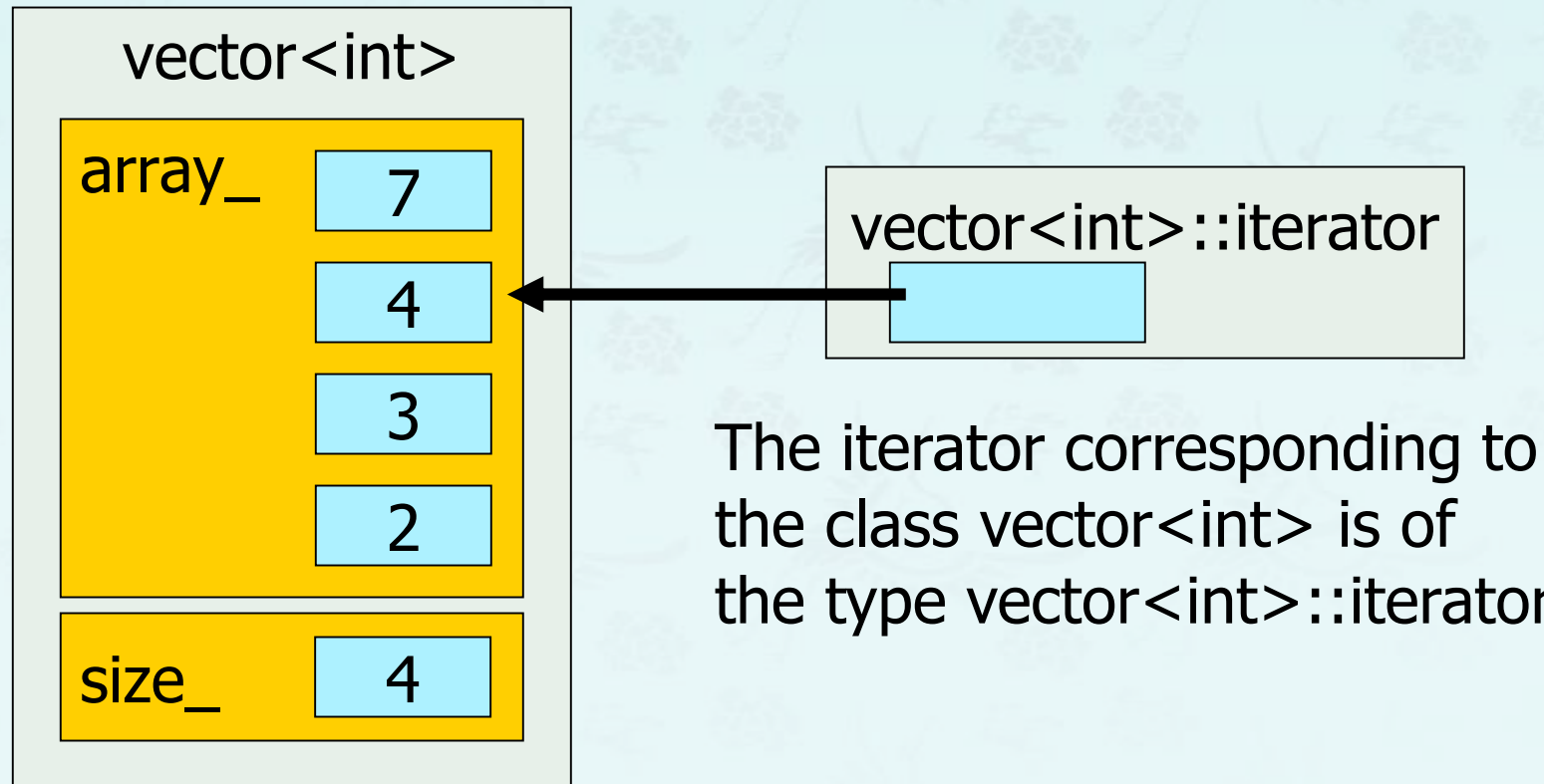
}
```

range-based for loop

range-based for loop

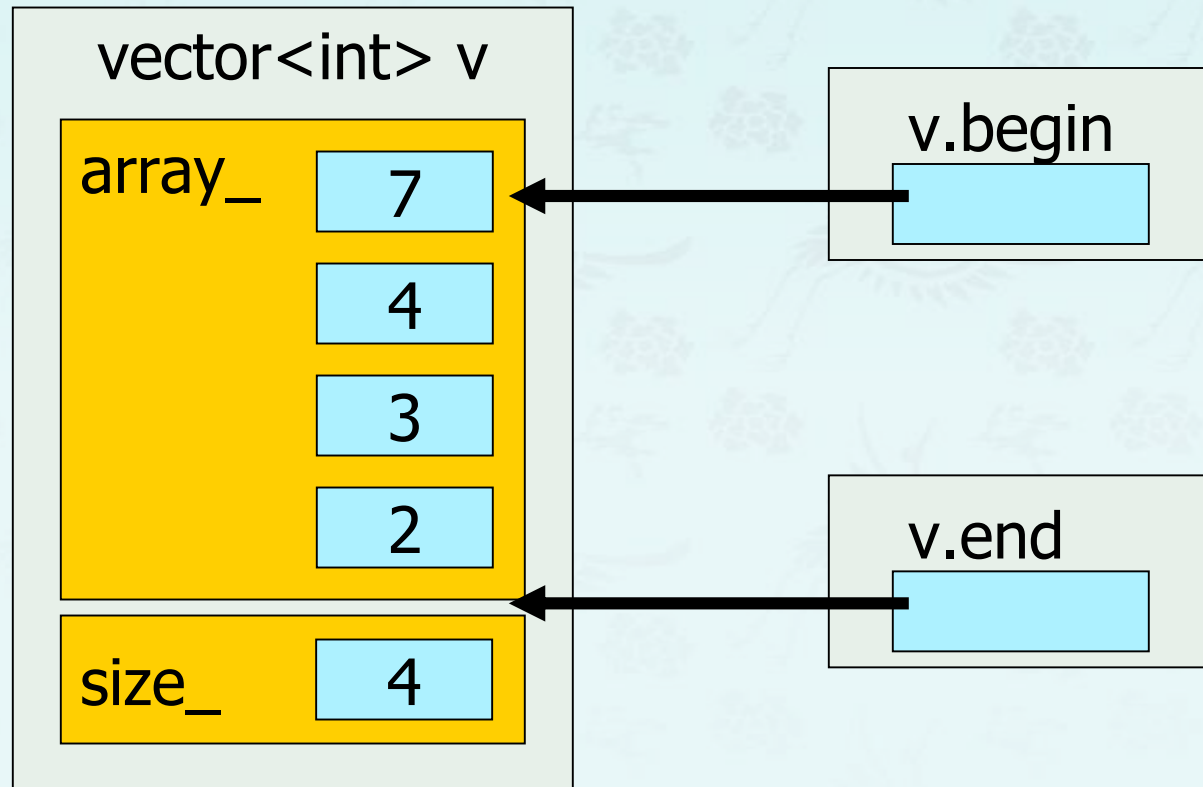
Iterators - 반복자

- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often they are used to move sequentially from element to element, a process called *iterating* through a container.



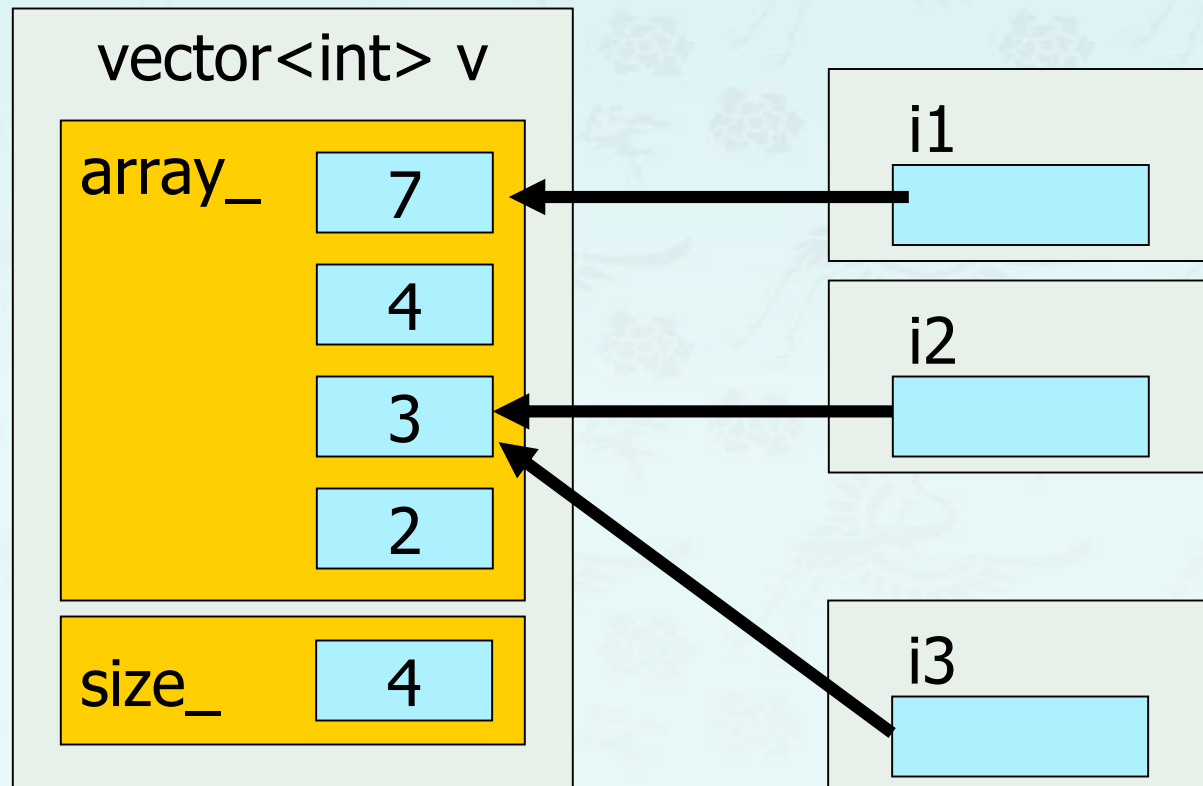
Iterators

- The member functions `begin()` and `end()` return an iterator to the first and **past the last element** of a container



Iterators

- One can have multiple iterators pointing to different or identical elements in the container



Iterators

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

int main() {
    int arr[] = { 7, 4, 3, 2 };           // standard C array
    vector<int> v(arr, arr+4);           // initialize vector with C array
    vector<int>::iterator it = v.begin(); // iterator for class vector
                                     ↖ auto

    // define iterator for vector and point it to first element of v
    cout << "1st element of v = " << *it; // de-reference iter
    it++;                                 // move iterator to next element
    it = v.end() - 1;                   // move iterator to last element
}
```

Iterators

```
int max(vector<int>::iterator start, vector<int>::iterator end) {  
    int m = *start;  
    while(start != end) {  
        if (*start > m) m = *start;  
        ++start;  
    }  
    return m;  
}
```

```
cout << "max of v = " << max(v.begin(), v.end());
```

Iterators

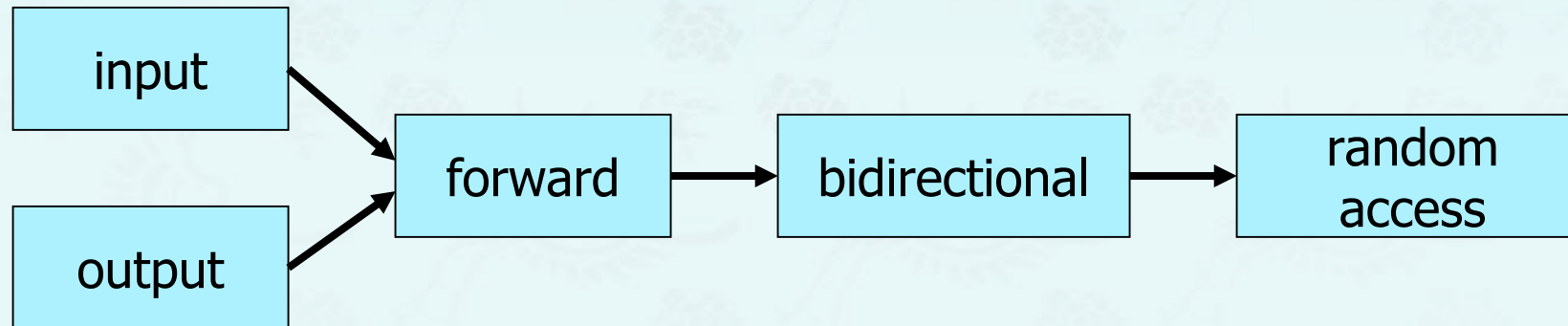
```
#include <vector>
#include <iostream>

int main() {
    int arr[] = { 7, 4, 3, 2 }; // standard C array
    vector<int> v(arr, arr+4); // initialize vector with C array

    for (auto i = v.begin(); i != v.end(); i++) {
        // initialize i with pointer to first element of v
        // i++ increment iterator, move iterator to next element
        cout << *i << " "; // de-referencing iterator returns the
                             // value of the element the iterator points at
    }
    cout << endl;
}
```

Iterator Categories

- Not every iterator can be used with every container for example the list class provides no random access iterator
- Every algorithm requires an iterator with a certain level of capability for example to use the **[] operator** you need a random access iterator
- Iterators are divided into five categories in which a higher (more specific) category always subsumes a lower (more general) category, e.g. An algorithm that accepts a forward iterator will also work with a bidirectional iterator and a random access iterator



Vector Container Example

`void push_back(const T& x);` - inserts an element with value x
at the end of the sequence.

`unsigned int size();` - returns the length of the sequence

Vector Container

```
int arr[5] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

3	7	9	11	8
---	---	---	----	---

Vector Container

```
int arr[5] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

3	7	9	11	8
---	---	---	----	---

Vector Container

```
int arr[5] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));
```

3	7	9	11	8
---	---	---	----	---

Only works during initialization



Vector Container

```
int arr[5] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

3	7	9	11	8
---	---	---	----	---

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + sizeof(arr)/sizeof(arr[0]));
```

Only works during initialization



Vector Container

```
int arr[5] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

3	7	9	11	8
---	---	---	----	---

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + 5);
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));
```

```
int arr[] = {3, 7, 9, 11, 8};  
vector<int> v(arr, arr + sizeof(arr)/sizeof(arr[0]));
```

```
vector<int> v{3, 7, 9, 11, 8};
```

Only works during initialization



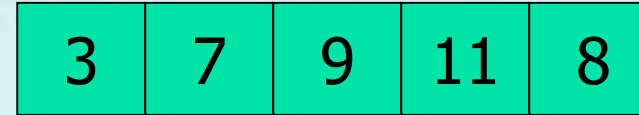
Vector Container

```
vector<int> v{3, 7, 9, 11, 8};
```

3	7	9	11	8
---	---	---	----	---

Vector Container

```
vector<int> v{3, 7, 9, 11, 8};
```



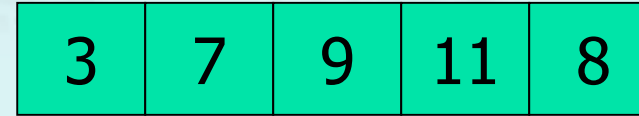
```
void foo(int *arr, int n) {  
    vector<int> v(arr, arr + n); ← Useful if arr is an argument in a function  
    vector<int> w{0, 1};  
    ...  
  
    // make w = [ w ] + [ v ] or push back w to w  


|  |  |  |
|--|--|--|
|  |  |  |
|--|--|--|

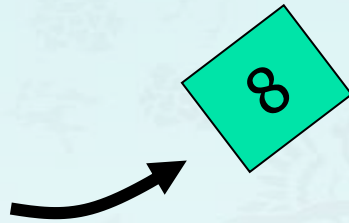
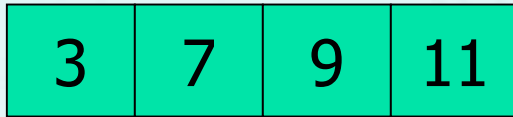
  
}
```


Vector Container

```
vector<int> v{3, 7, 9, 11, 8};
```

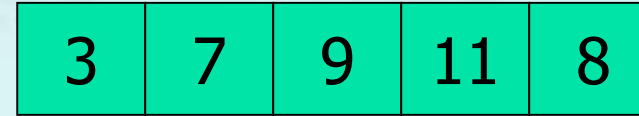


```
v.pop_back();
```

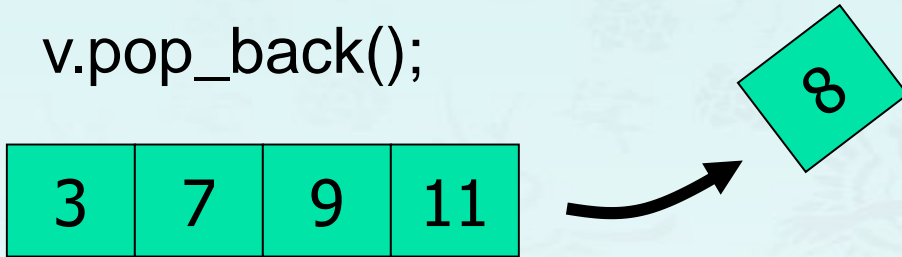


Vector Container

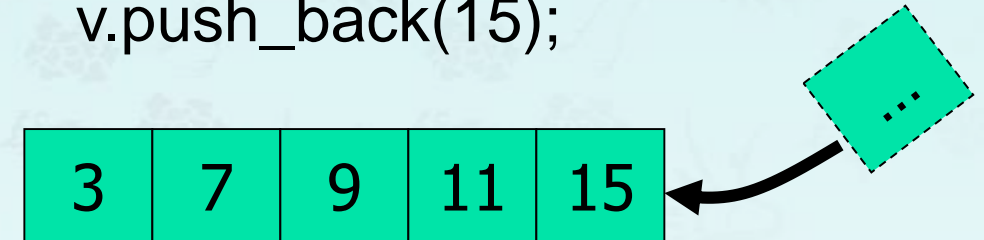
```
vector<int> v{3, 7, 9, 11, 8};
```



```
v.pop_back();
```

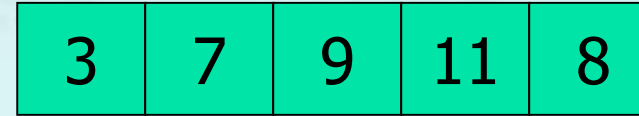


```
v.push_back(15);
```

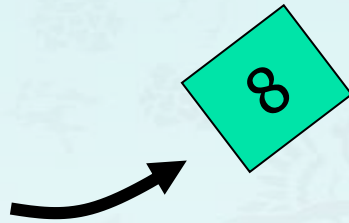
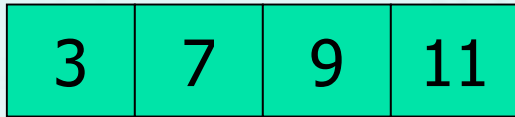


Vector Container

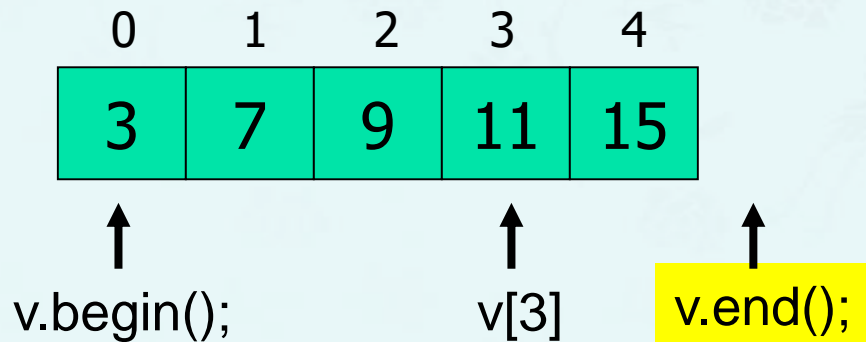
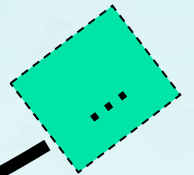
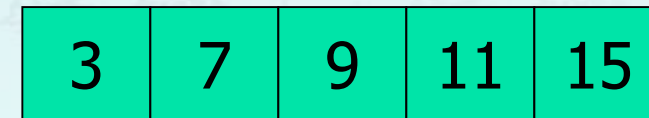
```
vector<int> v{3, 7, 9, 11, 8};
```



```
v.pop_back();
```

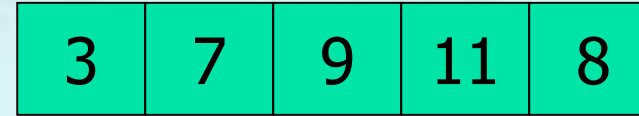


```
v.push_back(15);
```

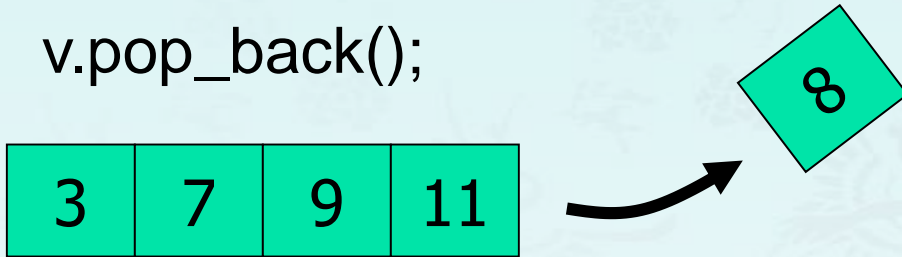


Vector Container

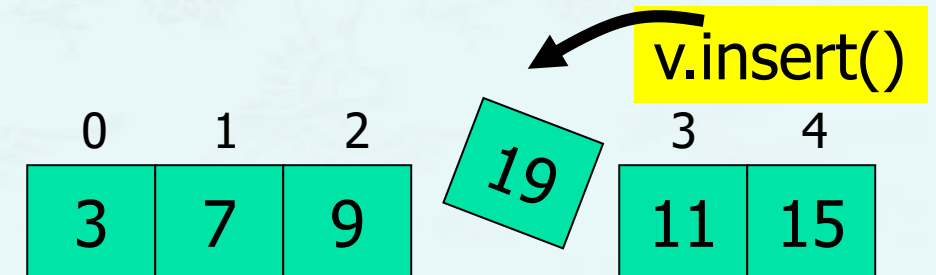
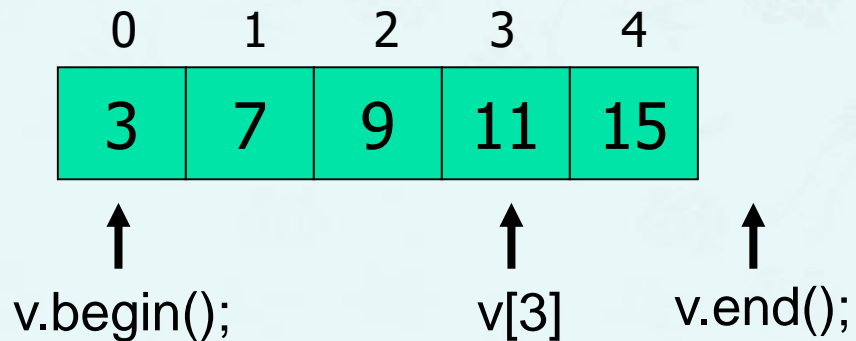
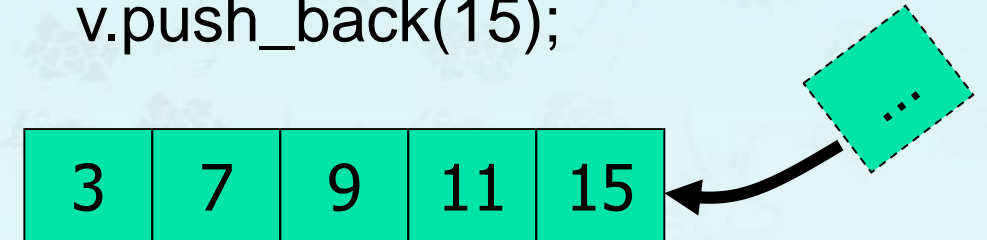
```
vector<int> v{3, 7, 9, 11, 8};
```



```
v.pop_back();
```

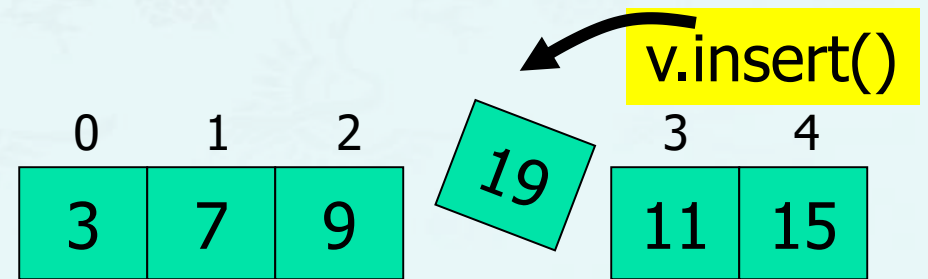
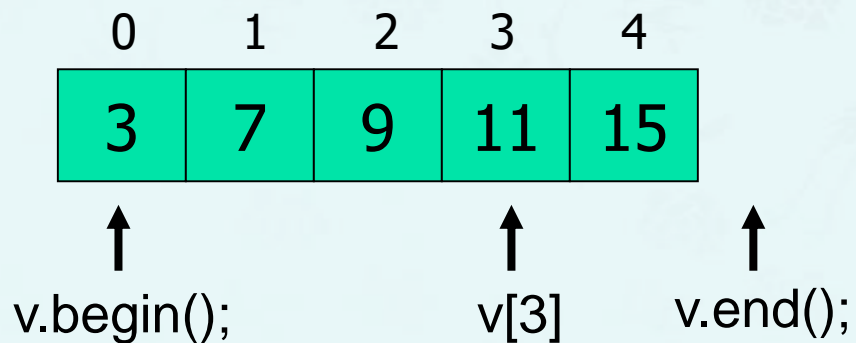


```
v.push_back(15);
```



Vector Container – insert()

```
int main() {  
    vector<int> vec{ 3, 7, 9, 11, 15 };  
  
    vec.insert( vec.begin() + 3, 19 );  
    for( auto x: vec )  
        cout << x << " ";  
    cout << endl;  
}
```



Vector Container – find()

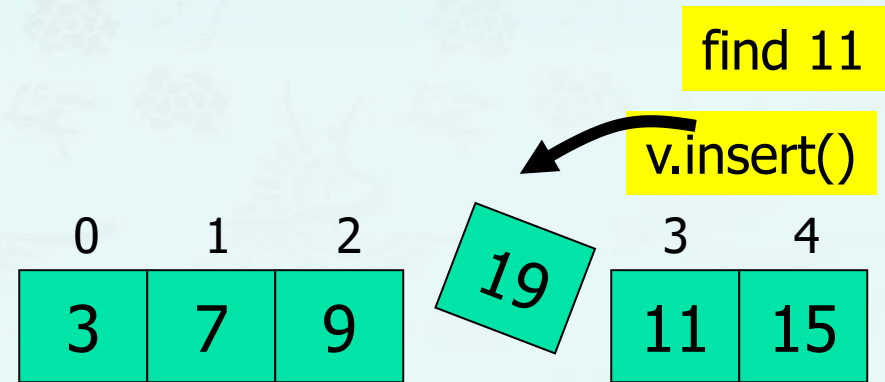
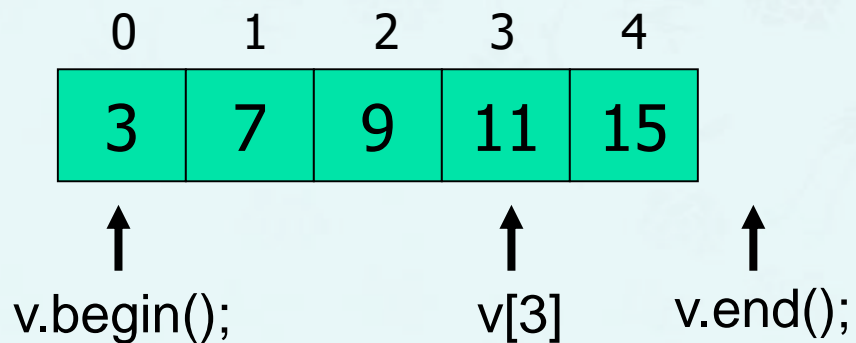


Vector Container – find()

```
#include <algorithm>
#include <vector>

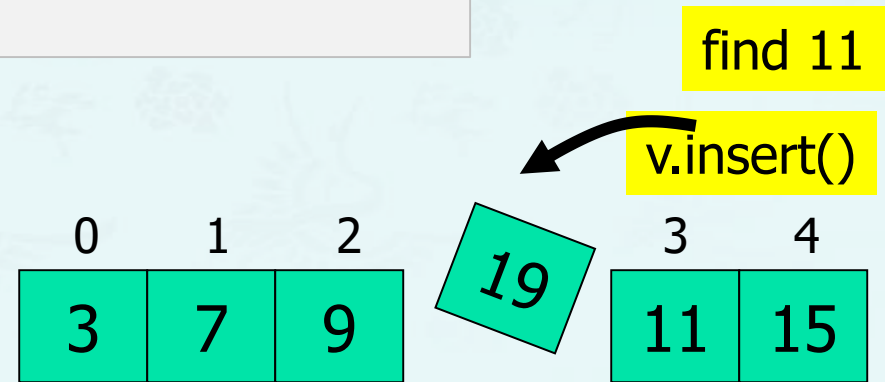
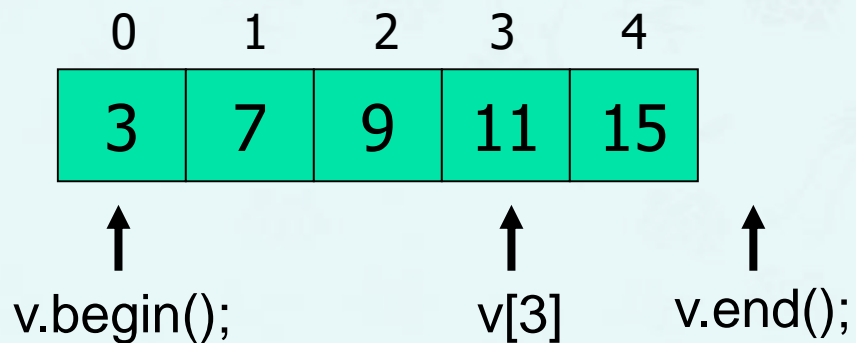
vector<int>::iterator it = find(vec.begin(), vec.end(), item)
if (it != vec.end() )
    do_this();
else
    do_that();
```

auto



Vector Container – find()

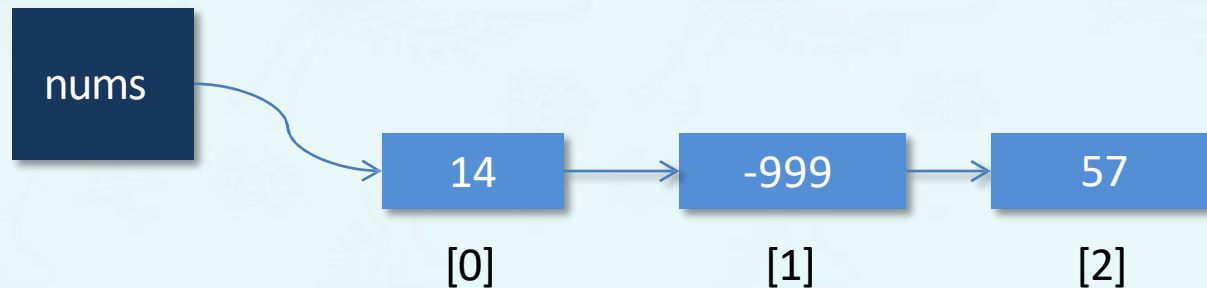
```
int main() {  
    vector<int> v{ 3, 7, 9, 11, 15 };  
  
    auto it = find( vec.begin(), vec.end(), 11 );  
    vec.insert( it, 19 );  
    for( auto x: vec )  
        cout << x << " ";  
}
```



Vector Container Example

```
#include <vector>
#include <iostream>
int main() {
    vector<int> nums; // create a vector of ints of size 3
    nums.insert(nums.begin(), -999); // -999
    nums.insert(nums.begin(), 14); // 14 -999
    nums.insert(nums.end(), 57); // 14 -999 57
    for (int i = 0; i < nums.size(); i++)
        cout << nums[i] << endl;
    nums.erase(nums.begin()); // -999 57
    nums.erase(nums.begin()); // 57
}
```

```
for ( auto x : nums )
    cout << x << endl;
```



Vector Container Exercise


- Print out vector object that has a member object as its first data.

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

class Member {
public:
    Member(string s, double d) : name(s), year(d) {}
    void print(); {
        cout << name << " " << year << endl;
    }
private:
    string name;
    double year;
};
```

Vector Container Exercise

- Print out vector object that has a member object as its first data.

```
int main() {  
    vector<Member> v;  
    v.push_back(Member("David", 15));  
    v.push_back(Member("Peter", 20));  
  
    vector<Member>::iterator  auto it = v.begin();  
    cout << "print all using iterator << endl;  
    while(it != v.end())  
        (it++)->print();  
    cout << endl;
```

```
// print all using for-loop.  
for(auto x : v)  
    x.print();  
cout << endl;  
  
cout << "checking the front()" << endl;  
v.front().print();  
return 0;  
}
```

Vector Container Exercise

- Write a program that reads integers from the user, sorts them, and print the result using
- (1) for each and
- (2) iterator.

```
int main() {
    int input;
    vector<int> vec;

    while (cin >> input )           // get input
        vec.push_back(input);

    sort(vec.begin(), vec.end());    // sorting

    vector<int>::iterator it;         // output
    for ( it = vec.begin(); it != vec.end(); ++it )
        cout << *it << " ";

    cout << endl;

    return 0;
}
```

Vector Container Exercise

- Write a program that reads integers from the user, sorts them, and print the result using
- (1) for each and
- (2) iterator.

```
int main() {  
    int input;  
    vector<int> vec;  
  
    while (cin >> input )           // get input  
        vec.push_back(input);  
  
    sort(vec.begin(), vec.end());    // sorting  
  
    for ( auto = vec.begin(); it != vec.end(); ++it )  
        cout << *it << " ";  
  
    cout << endl;  
  
    return 0;  
}
```

For_Each() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
void show_sqr(int n) {
    cout << n * n << " ";
}

int arr[] = { 7, 4, 3, 2 };           // standard C array
vector<int> v(arr, arr+4);           // initialize vector with C array
for_each (v.begin(), v.end(), show_sqr); // apply function show
                                           // to each element of vector v
```

Find_If() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
bool mytest(int n) { return (n > 2) && (n < 7); };

int main() {
    int arr[] = { 2, 3, 7, 8, 4, 6, 9 };           // standard C array
    vector<int> v(arr, arr+7);                     // initialize vector with C array

    auto iter = find_if(v.begin(), v.end(), mytest);
    if (iter != v.end())
        cout << "found " << *iter << endl;
    else
        cout << "not found" << endl;
}
```


Count_If() Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
bool mytest(int n) { return (n > 2) && (n < 7); };

int main() {
    int arr[] = { 2, 3, 7, 8, 4, 6, 9 };           // standard C array
    vector<int> v(arr, arr+7);                     // initialize vector with C array

    int n = count_if(v.begin(), v.end(), mytest);

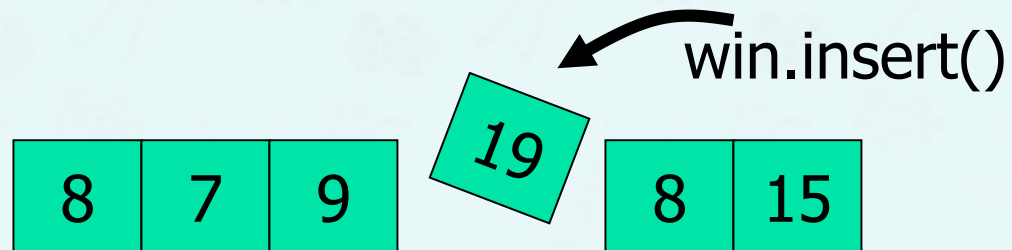
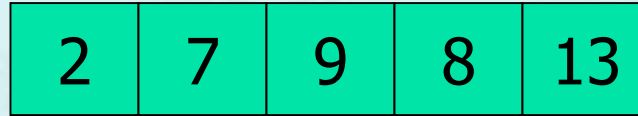
    // counts element in v for which mytest() is true
    cout << "found " << n << " elements" << endl;
}
```

List Container

- An STL list container is a double linked list, in which each element contains a pointer to its successor and predecessor.
- It is possible to add and remove elements from both ends of the list.
- Lists do not allow random access but are efficient to insert new elements and to sort and merge lists.

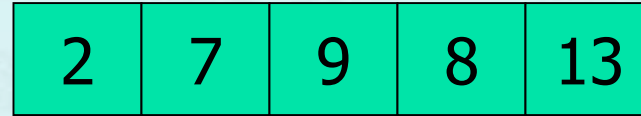
List Container

```
int arr[] = {2, 7, 9, 8, 13};  
list<int> win(arr, arr+5);
```

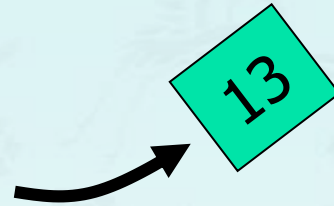
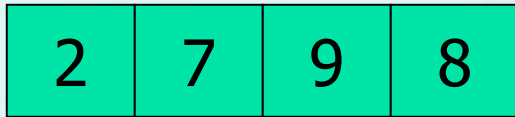


List Container

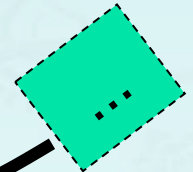
```
list<int> win{ 2, 7, 9, 8, 13 };
```



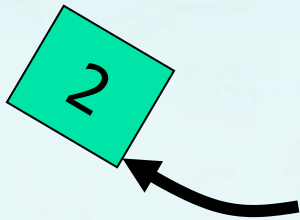
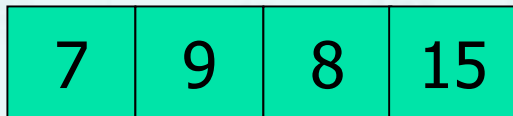
win.pop_back();



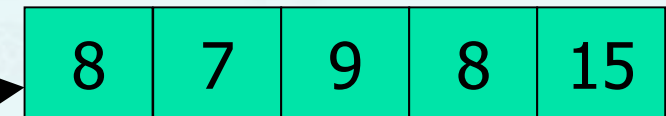
win.push_back(15);



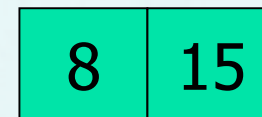
win.pop_front();



win.push_front(8);

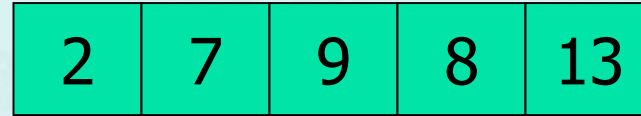


win.insert()

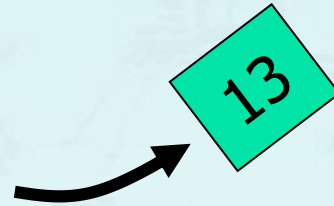
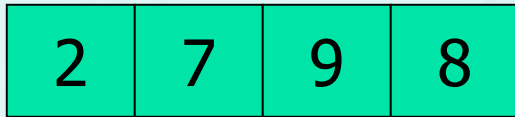


List Container

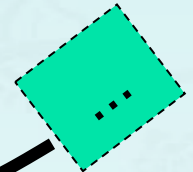
```
list<int> win{ 2, 7, 9, 8, 13 };
```



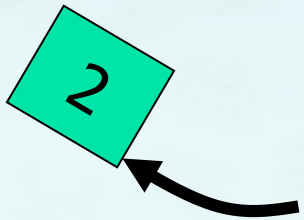
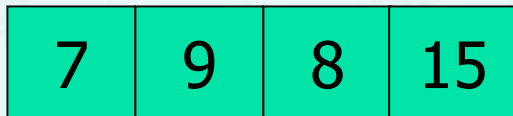
win.pop_back();



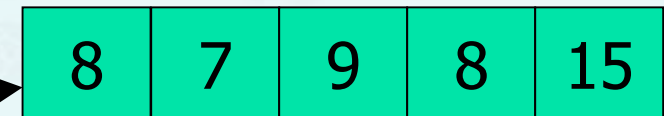
win.push_back(15);



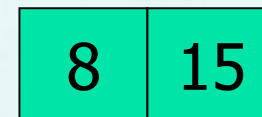
win.pop_front();



win.push_front(8);



win.insert()



List example – find_end()

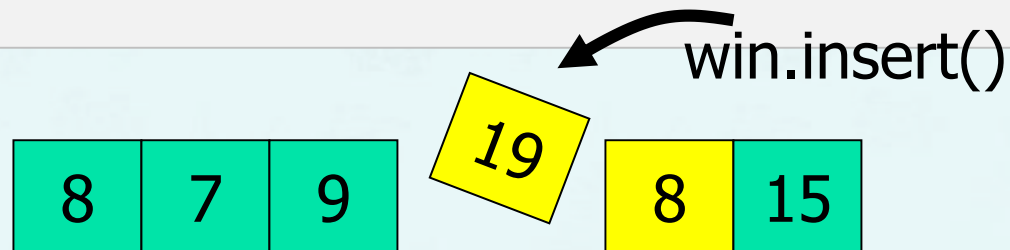
- If you normally copy elements using the copy algorithm you overwrite the existing contents

```
#include <list>

int main() {
    list<int> win{ 8, 7, 9, 8, 13 };
    for (auto i: win) cout << i << " "; cout << endl;

    list<int> ins{8};
    auto it = find_end(win.begin(), win.end(), ins.begin(), ins.end());
    win.insert(it, 19);

    for (auto x: win) cout << x << " "; cout << endl;
}
```



List example

- If you normally copy elements using the copy algorithm you overwrite the existing contents

```
#include <list>

int main() {
    list<string> names;
    names.insert(names.begin(), "Eric");
    names.insert(names.end(), "John");
    names.insert(names.end(), "Daniel");
    cout << *(names.begin()) << endl;
    names.reverse();
}
```



List exercise

- Create a member class and two lists. Combine two lists into one and remove duplicate.

```
#include <list>
#include <algorithm>
using namespace std;

class Member {
public:
    Member(string f, string l) : first_n(f), last_n(l) {}
    void print() {
        cout << last_n << " " << first_n << endl;
    }
private:
    string last_n, first_n;
    friend bool operator < (Member& m1, Member& m2) { return m1.last_n < m2.last_n; }
    friend bool operator == (Member& m1, Member& m2) { return m1.last_n == m2.last_n; }
};
```


List exercise

- Create a member class and two lists. Combine two lists into one and remove duplicate.


```
int main () {
    list<Member> li1;
    li1.push_back(Member("Linda","Smith"));
    li1.push_back(Member("Robert","Frost"));
    li1.push_back(Member("Alex","Amstrong"));

    list<Member> li2;
    li2.push_back(Member("Linda","Smith"));
    li2.push_back(Member("John","Wood"));
    li2.push_back(Member("Alex","Amstrong"));
```

```
    li1.sort();
    li2.sort();
    li1.merge(li2);

    cout << "li1 after sorting and merge" << endl;
    list<Member>::iterator it = li1.begin();
    while ( it != li1.end() )
        (it++)->print();
    li1.unique();
    cout << "After li1.unique()" << endl;

    it = li1.begin();
    while ( it != li1.end() )
        (it++)->print();
    return 0;
}
```



Insert Iterators

- If you normally copy elements using the copy algorithm you overwrite the existing contents

```
#include <list>
int arr1[]= { 1, 3, 5, 7, 9 };
int arr2[]= { 2, 4, 6, 8, 10 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2

copy(l1.begin(), l1.end(), l2.begin());
// copy contents of l1 to l2 overwriting the elements in l2
// l2 = { 1, 3, 5, 7, 9 }
```

Sort & Merge

- Sort and merge allow you to sort and merge elements in a container

```
#include <list>
int arr1[]= { 6, 4, 9, 1, 7 };
int arr2[]= { 4, 2, 1, 3, 8 };
list<int>  l1(arr1, arr1+5); // initialize l1 with arr1
list<int>  l2(arr2, arr2+5); // initialize l2 with arr2

l1.sort(); // l1 = {1, 4, 6, 7, 9}
l2.sort(); // l2= {1, 2, 3, 4, 8 }

l1.merge(l2); // merges l2 into l1
// l1 = { 1, 1, 2, 3, 4, 4, 6, 7, 8, 9},  l2= {}
```

Associative Containers

- Why Associative Containers?
 - Map
 - Pair
 - Copy algorithm

Associative Containers

- In an associative container the items are not arranged in sequence, but usually as a tree structure or a hash table.
- The main advantage of associative containers is **the speed of searching** (binary search like in a dictionary)
- Searching is done using a **key** which is usually a single value like a number or string
- The **value** is an attribute of the objects in the container
- The STL contains two basic associative containers
 - **sets** and multisets
 - **maps** and multimaps

Why Associative Containers?

- Let us suppose that we all employee data are saved in a vector.

```
class Employee {
public:
    // Constructors ...:
    Employee () {}
    Employee (const string& name) : name(n) {}

    // Member functions ....:
    void set_year(double y) { year = y; }
    double year() const { return year; }
    void set_name(const string& n) { name = n; }
    const string& name() const { return name;}
    // ...
private:
    double   year;
    string   name;
};
```

Why Associative Containers?

- When we need to find a specific employee:
 - go over all employees until you find one that its name matches the requested name.
 - **Bad solution** - not efficient!

Solution:

Map – Associative Array

Why Associative Containers?

- Solution:
Map – Associative Array
- Most useful when we want to store (and possibly modify) an associated value.
- We provide a **key/value pair**. The key serves as an index into the map, the value serves as the data to be stored.
- Insertion/find operation - $O(\log n)$
- Have a map, where the **key** will be the employee **name** and the **value** – the employee object.
 - name → employee.
 - string → class Employee
- `map<string, Employee*> employees;`

Populating a Map and locating an employee

```
// populating map
void main() {
    map<string, Employee*> employees;

    string name("Eti");
    Employee* employee;
    employee = new Employee(name);

    //insertion
    employees[name] = employee;
}
```

```
// locating employee
map<string, Employee*> employees;

// Looking for an employee named Eti :
//find
Employee *eti = employees["Eti"];

//or
map<string, Employee *>::iterator iter =
    employees.find("Eti");
```

The returned value is an iterator to map.
If "Eti" exists on map,
it points to this value,
otherwise,
it returns the end() iterator of map.

Iterating a Map

```
// Printing all map contents.
```

```
map<string, Employee*>::iterator it;  
for ( it = employees.begin(); it != employees.end(); ++it )  
{  
    cout << ???  
}
```

Pointer Semantics

- Let **iter** be an iterator then :
 - **++iter** (or `iter++`) advances the iterator to the next element
 - ***iter** returns the value of the element addressed by the iterator.
- Each container provide a `begin()` and `end()` member functions.
- **begin()** Returns an iterator that addresses the first element of the container.
- **end()** returns an iterator that addresses 1 past the last element.

Iterating Over Containers

- Iterating over the elements of any container type.
- ```
for (iter = container.begin(); iter != container.end(); ++iter)
{
 // do something with the element
}
```

## Map Iterators

---

- `map<key, value>::iterator iter;`
- What type of element `iter` does addresses?
  - The **key** ?
  - The **value** ?
- It addresses a **key/value** pair.

## Pair

---

- Stores a pair of objects, first of type T1, and second of type T2.
- ```
struct pair<T1, T2>  
{  
    T1 first;  
    T2 second;  
};
```

■

Our Pair

- In our example **iter** addresses a pair **<string, Employee *>** element.
- Accessing the name (key)
iter→first
- Accessing the Employee* (value)
iter→second

For example: Printing the Salary

```
for ( iter = employees.begin(); iter != employees.end(); ++iter )  
{  
    cout << iter->first << " " << (iter->second)->salary();  
}
```

Map Sorting Scheme

- map holds its content **sorted by key**.

Map Sorting Problem:

- If we want to sort the map using another sorting scheme, for example, by salary, what should we do?
 - **Problem:**
Since map already holds the elements sorted, we can't sort them.
 - **Solution:**
Copy the elements to a container where we can control the sorting scheme.

Copy

- `copy(iterator first, iterator last, iterator where);`
- Copy from 'first' to 'last' into 'where'.

```
int ia[] = { 0, 1, 1, 2, 3, 5, 5, 8 };  
vector<int> vec1(ia, ia + 8), vec2;  
// ...  
copy( vec1.begin(), vec1.end(), vec2.begin() );
```

The Problem:

- `vec2` has been allocated no space.
- The copy algorithm uses assignment to copy each element value.
- `copy` will fail, because there is no space available.

The Solution: use `back_inserter()`

- Causes the container's **`push_back`** operation to be invoked.
- The argument to **`back_inserter`** is the container itself.

The Solution:

- ```
// ok. copy now inserts using vec2.push_back()
copy(vec1.begin(), vec1.end(), back_inserter(vec2));
```

## Insertter iterators.

- Puts an algorithm into an "insert mode" rather than "over write mode".



- \*iter = causes an insertion at that position, (instead of overwriting).

**Now, Employee copy works.**

```
map<string, Employee *> employees;
vector< pair<string, Employee*> > evec;

copy(employees.begin(), employees.end(), back_inserter(evec));
```

# Sort

---

- Formal definition :

```
void sort(Iterator first, Iterator last);
```

- Example:

```
vector<int> vec;
// Fill vec with integers ...
sort(vec.begin(), vec.end())
```

# Sort

---

- Sort uses operator < to sort two elements.
- What happens when sorting is meaningful, but no operator < is defined ?

- **The meaning of operator <**

What does it mean to write :

```
pair<string, Employee*> p1, p2;
if (p1 < p2) {
 ...
}
```

- No operator < is defined between two pairs.
- How can we sort a vector of pairs ?

## Sorting Function

- Define a function that knows how to sort these elements, and make the sort algorithm use it.
- **lessThen Function**

```
bool lessThen(pair<string, Employee *> &l, pair<string, Employee *> &r)
{
 return (l.second)->Salary() < (r.second)->Salary()
}
```

## Using lessThen Function

---

- `vector< pair<string, Employee *> > evec;`
- `// Use lessThen to sort the vector.`  
`sort(evec.begin(), evec.end(), lessThen);`

pointer to function



## Putting it all Together

```
bool lessThen(pair<...> &p1, pair<...> &p2) { ... }

int main() {
 map<string, Employee *> employees;

 // Populate the map.
 vector< pair<string, Employee *> > employeeVec;

 copy(employees.begin(), employees.end(), back_inserter(employeeVec));

 sort(employeeVec.begin(), employeeVec.end(), lessThen);

 vector< pair<string, Employee *> >::iterator it;
 for (it = employeeVec.begin(); it != employeeVec.end(); ++it) {
 cout << (it->second)->getName() << " " << (it->second)->getSalary() << endl;
 }
 return 0;
}
```



## Sets and Multisets

```
#include <set>
string names[] = {"Ole", "Hedvig", "Juan", "Lars", "Guido"};
set<string, less<string> > nameSet(names, names+5);

// create a set of names in which elements are alphabetically
// ordered string is the key and the object itself
nameSet.insert("Patric"); // inserts more names
nameSet.insert("Maria");
nameSet.erase("Juan"); // removes an element

set<string, less<string> >::iterator iter; // set iterator
string searchname;
cin >> searchname;
iter=nameSet.find(searchname); // find matching name in set
if (iter == nameSet.end()) // check if iterator points to end of set
 cout << searchname << " not in set!" <<endl;
else
 cout << searchname << " is in set!" <<endl;
```

## Sets and Multisets

```
string names[] = {"Ole", "Hedvig", "Juan", "Lars", "Guido", "Patric", "Maria", "Ann"};
set<string, less<string> > nameSet(names, names+7);
set<string, less<string> >::iterator iter; // set iterator

iter = nameSet.lower_bound("K");
// set iterator to lower start value "K"
while (iter != nameSet.upper_bound("Q"))
 cout << *iter++ << endl;

// displays Lars, Maria, Ole, Patric
```

## Maps and Multimaps

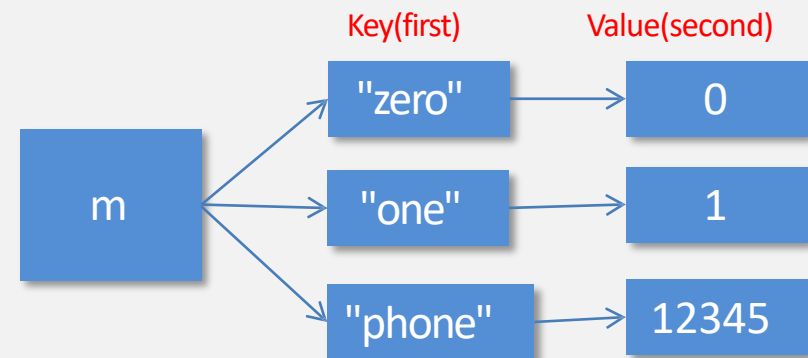
---

- A map stores pairs **<key, value>** of a key object and associated value object.
- The key object contains a key that will be searched for and the value object contains additional data
- The key could be a string, for example the name of a person and the value could be a number, for example the telephone number of a person

# Map Container Example

```
#include <map>

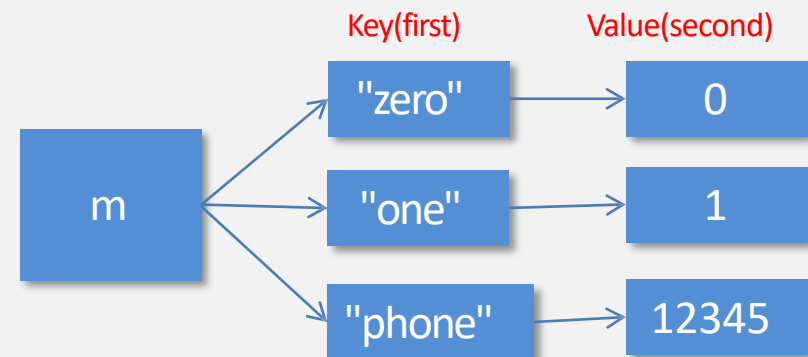
map<string, int> m;
// key and value data types are string and int
m["zero"] = 0;
m["one"] = 1;
m["phone"] = 12345;
cout << m["one"] << m["phone"] << endl;
// 112345
```



# Map Container Example

```
#include <map>

map<string, int> m;
// key and value data types are string and int
m["zero"] = 0;
m["one"] = 1;
m["phone"] = 12345;
map<string, int>::iterator it = m.begin();
while(it != m.end()) {
 cout << (*it).first << " "
 << (*it).second << endl;
 it++;
}
```



## Map Container Exercise

- Create a <word, meaning> pair map as a dictionary, then get an input of a word from the user and print its meaning.

```
#include <map>
int main () {
 map<string, string> dictionary;
 string searchWord;
 //(1) operator [] 를 이용한 입력 : overwrite 가능
 dictionary["horse"] = "말"; dictionary["apple"] = "사과";
 //(2) insert 메소드를 이용한 입력 : overwrite 불가

 dictionary.insert(pair<string, string>("grape", "포도"));
 dictionary.insert(pair<string, string>("orange", "오렌지"));
 // 영어 단어를 이용한 검색
 cout << "검색하고자 하는 영어 단어를 입력하세요 : ";
 cin >> searchWord;
 if(!dictionary[searchWord].empty()) {
 cout << "단어를 찾았습니다." << endl;
 cout << searchWord << " : " << dictionary[searchWord] << endl;
 }
 else {
 cout << "검색된 단어가 없습니다." << endl;
 }
 return 0;
}
```

※동일하게 동작하도록 first, second를 이용하여 구현.  
(이전 페이지 참조)

```
map<string, string>::iterator it;
it = dictionary.find(searchWord);
if(!(*it).second.empty()) {
 cout << "단어를 찾았습니다." << endl;
 cout << __ << " : " << __ << endl;
}
else {
 cout << "검색된 단어가 없습니다." << endl;
}
```

## Maps and Multimaps

```
#include <map>
string names[]= {"Ole", "Hedvig", "Juan", "Lars", "Guido", "Patric", "Maria", "Ann"};
int numbers[]= {75643, 83268, 97353, 87353, 19988, 76455, 77443,12221};

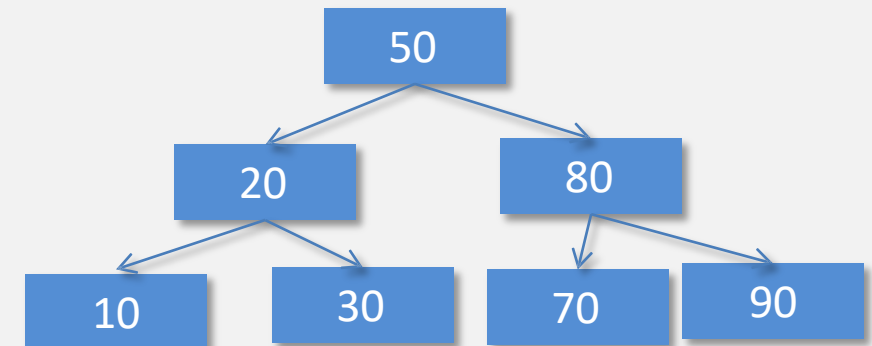
map<string, int, less<string> > phonebook;
map<string, int, less<string> >::iterator iter;

for (int j=0; j<8; j++)
 phonebook[names[j]]=numbers[j]; // initialize map phonebook

for (iter = phonebook.begin(); iter !=phonebook.end(); iter++)
 cout << (*iter).first << " : " << (*iter).second << endl;
cout << "Lars phone number is " << phonebook["Lars"] << endl;
```

## Set Container Example

```
#include <set>
s.insert(50);
s.insert(20);
s.insert(10);
s.insert(80);
s.insert(30);
s.insert(70);
s.insert(90);
// set<int>::iterator iter;
for (auto iter = s.begin(); iter != s.end(); iter++)
 cout << *iter << ' ';
```





## Set Container Exercise

- Create a set that has Member objects and search "Frost Robert".

```
class Member {
public:
 Member(string l, string f) : last(l), first(f){} void print() const{
 cout.setf(ios::left);
 cout << setw(15) << first << " " << last << endl;
 }
private:
 string first, last;
 friend bool operator < (const Member& m1, const Member& m2) {
 return (m1.last < m2.last) ? true : false;
 }

 friend bool operator == (const Member& m1, const Member& m2) {
 return (m1.last == m2.last) ? true : false;
 }
}
```

## Set Container Exercise

- Create a set that has Member objects and search "Frost Robert".

```
int main () {
 typedef Member M;
 typedef set<M> S;
 M m("Frost","Robert");

 S s; s.insert(m);
 s.insert(M("Smith","John"));
 s.insert(M("Amstrong","Bill"));
 s.insert(M("Bain","Linda"));
 s.insert(M("Amstrong","Bill")); //두 번째 입력과 동일

 // 기존에 존재하는 값들과 비교하여, 동일한 값이 이미 존재 할 경우 값을 추가하지 않는다.
 // 이 동작은 operator ==를 통해 이루어진다.

 S::iterator it = s.begin();
 while (it != s.end()) {
 (it++)->print(); it = s.find(m);
 if (it == s.end())
 cout << "element not found" << endl;
 else {
 cout << "element is found : "; (*it).print();
 }
 }
 return 0;
}
```

## Use binary search algorithm

- Create a vector with int data, then use std::sort to sort.

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>
using namespace std;
void main() {
 vector<int> v; v.push_back(10);
 v.push_back(20); v.push_back(30);
 v.push_back(40); v.push_back(50);

 if(binary_search(v.begin(), v.end(), 20))
 cout << "20 있음!" << endl;
 else
 cout << "20 없음!" << endl;

 if(binary_search(v.begin(), v.end(), 15))
 cout << "15 있음!" << endl;
 else
 cout << "15 없음!" << endl;
}
```



Summary

&

quaestio quaestio qo → q ? ? ?