STL

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

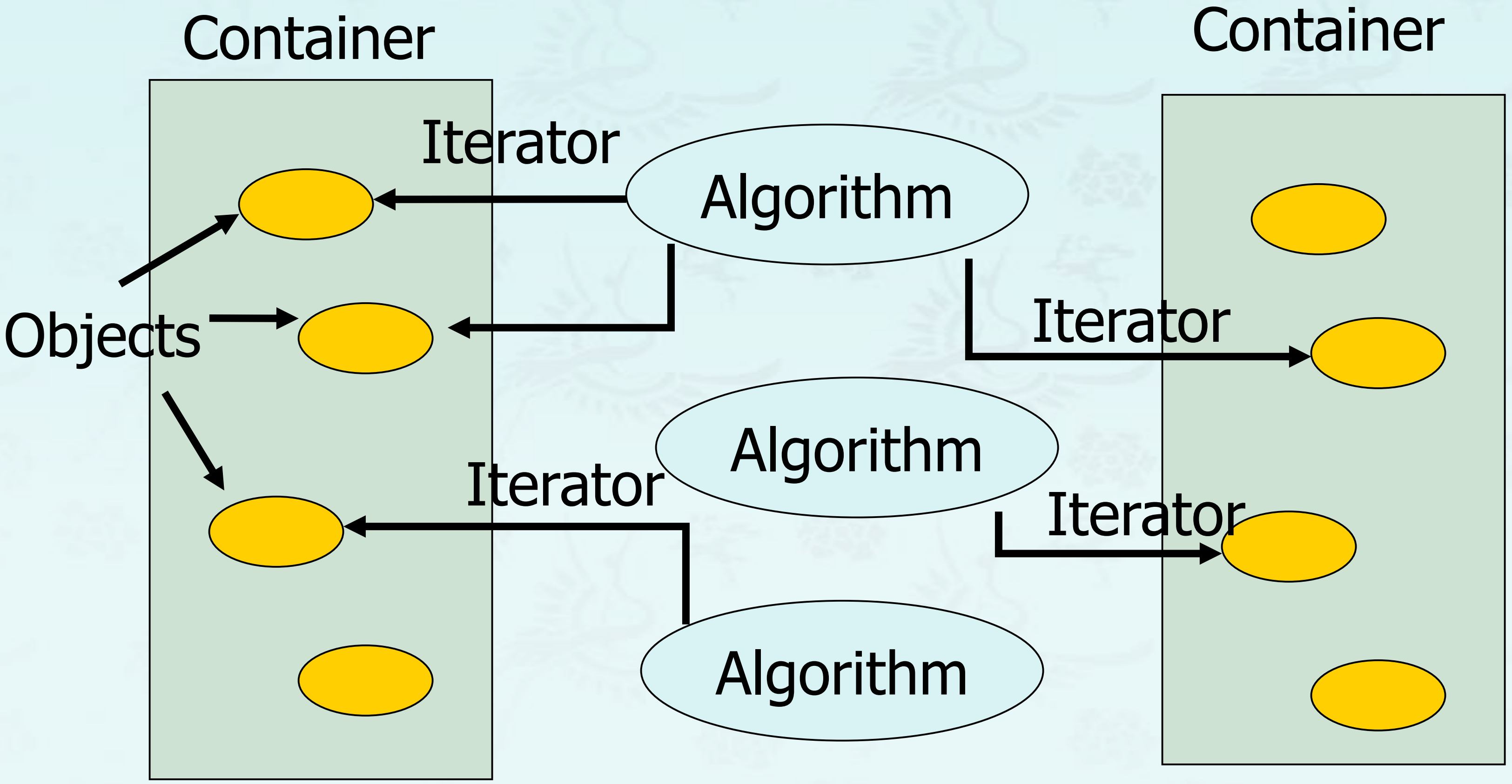
Standard Template Library

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	- 동적 배열이므로 배열의 크기를 변경할 수 있다. - 임의 접근이 가능하며, 뒤에서의 삽입이 빠르다.
list	- 연결 리스트이므로 데이터를 순차적으로 접근하고 관리할 때 유용하다. - 위치에 상관없이 삽입과 삭제가 빠르다.
deque	- 데크라고 한다. - 임의 접근이 가능하며, 앞과 뒤에서의 삽입이 빠르다.
map	- 특정 키(key)에 의해서 데이터를 접근하고 관리할 수 있다 - 키를 통해 값을 접근하며, 삽입과 삭제가 빠르다.
set	- 원소들을 순서대로 관리하며, 소속 검사와 삽입, 삭제가 빠르다. - 중복된 원소를 허용하지 않는다.
stack	- top에서만 삽입과 삭제가 가능하다. - LIFO(Last In First Out) 방식으로 데이터를 삽입, 삭제 한다.
queue	- 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

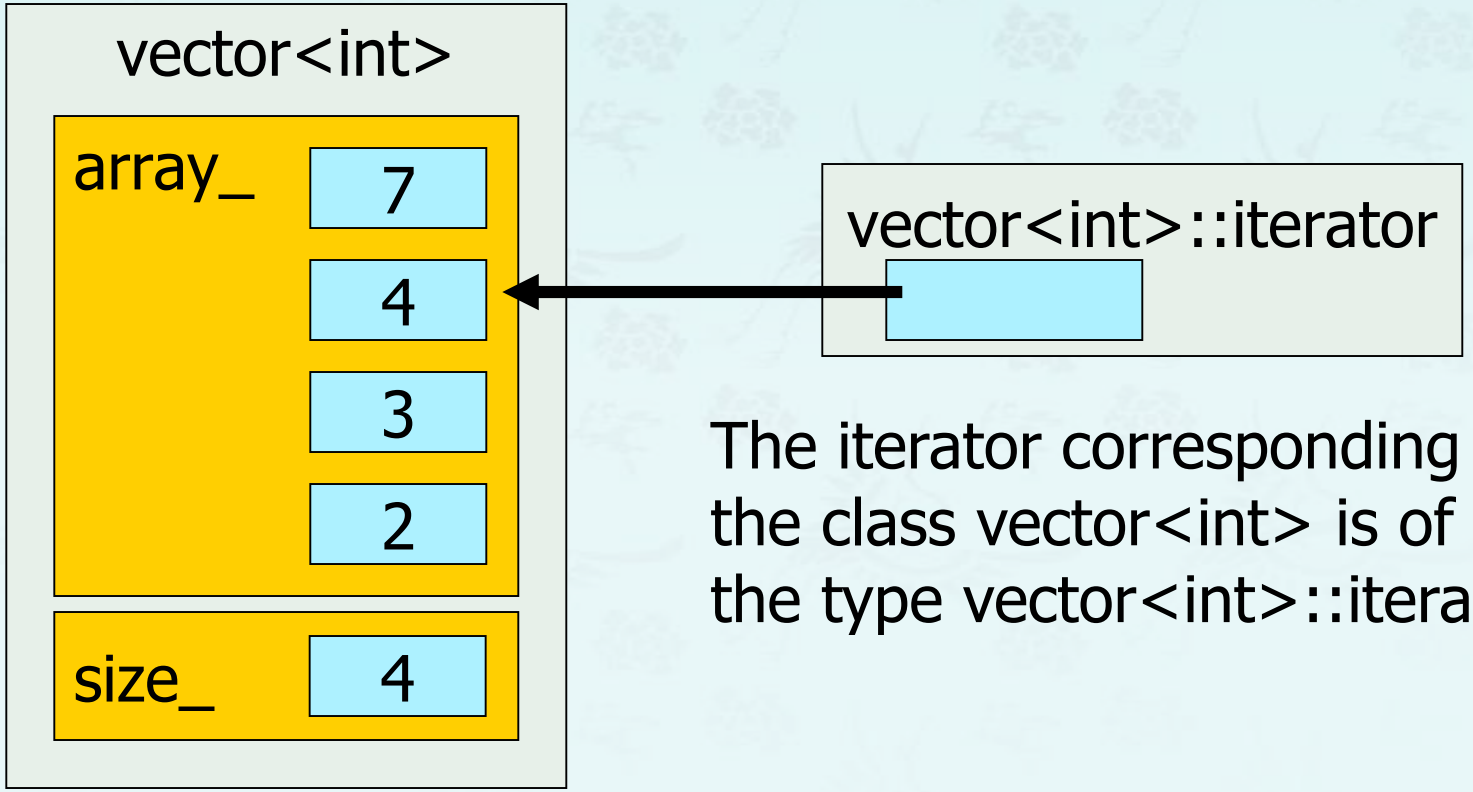
```
#include<iostream>
#include<vector>
using namespace std;
int main(){
  vector<int> v(5);
  for(int i=0; i < v.size(); i++)
    cin >> v[i];



}
```

range-based for loop

range-based for loop

Iterators - 반복자

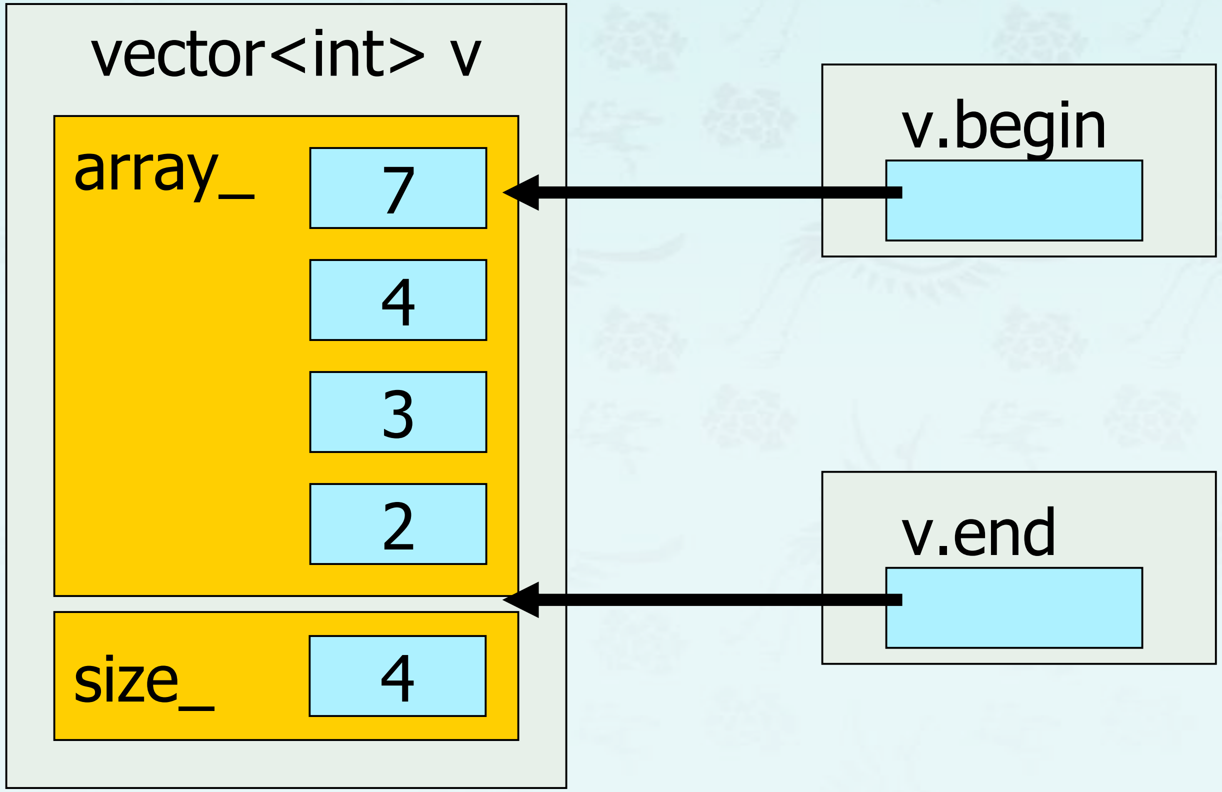
- Iterators are pointer-like entities that are used to access individual elements in a container.
- Often they are used to move sequentially from element to element, a process called *iterating* through a container.



The iterator corresponding to the class vector<int> is of the type vector<int>::iterator

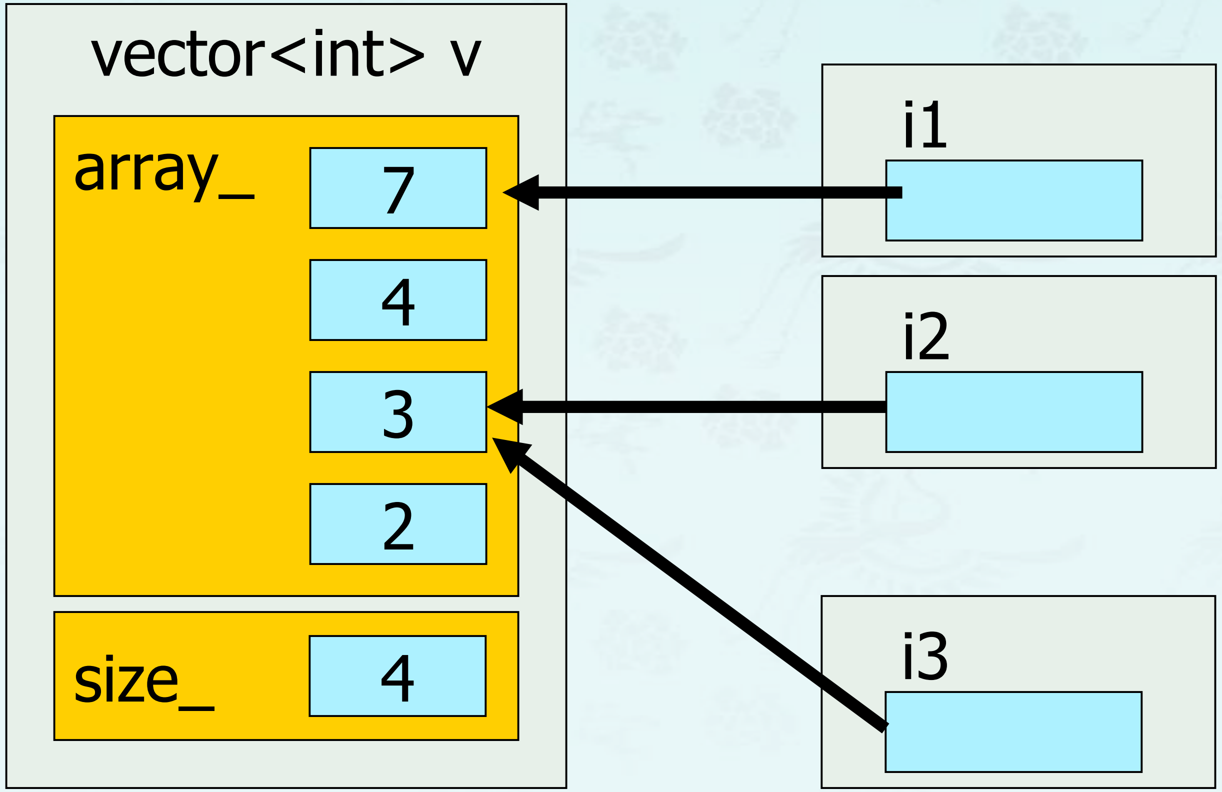
Iterators

- 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

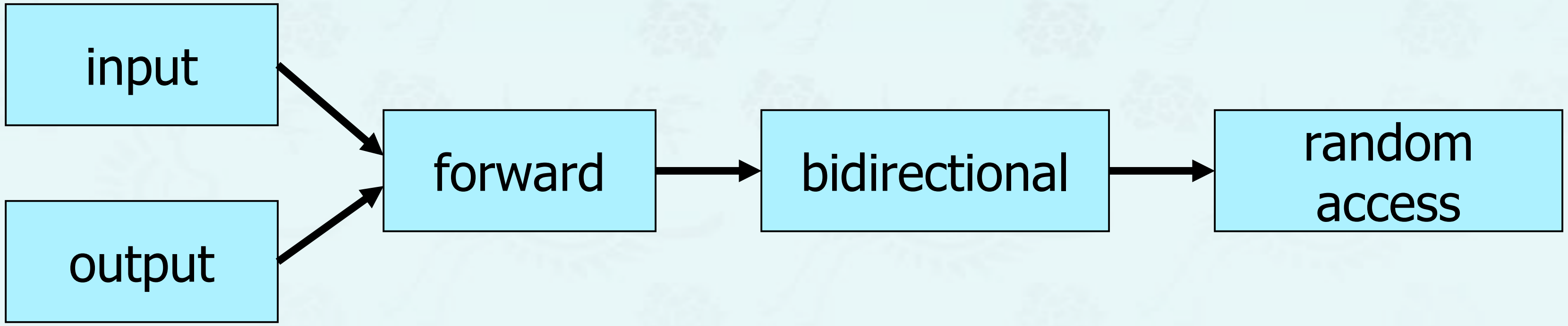
```cpp
int max(vector<int>::iterator start, vector<int>::iterator end) {
    int m = *start;
    while(start != end) {
      if (*start > m) m = *start;
      ++start;
    }
    return m;
}

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

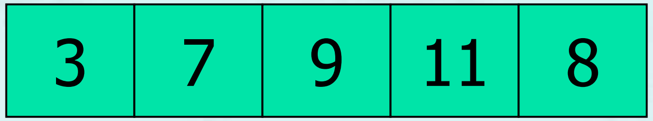


Vector Container Example

```
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);
```



Vector Container

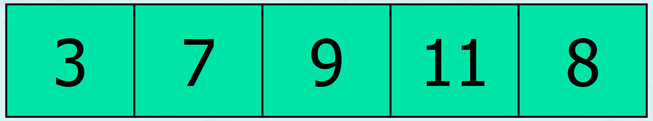
```
int arr[5] = {3, 7, 9, 11, 8};
vector<int> v(arr, arr + 5);
```

3	7	9	11	8

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

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);
```

Only works during initialization

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

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);
```

Only works during initialization

```
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 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);
```

Only works during initialization

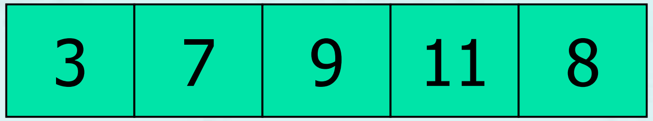
```
int arr[] = {3, 7, 9, 11, 8};
vector<int> v(arr, arr + sizeof(arr)/sizeof(int));
```

```
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};
```

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};
```

3	7	9	11	8

```
void foo(int *arr, int n) {
  vector<int> v(arr, arr + n);
  vector<int> w{0, 1};
  ...

  // make w = [ w ] + [ v ] or push back w to w

}
```

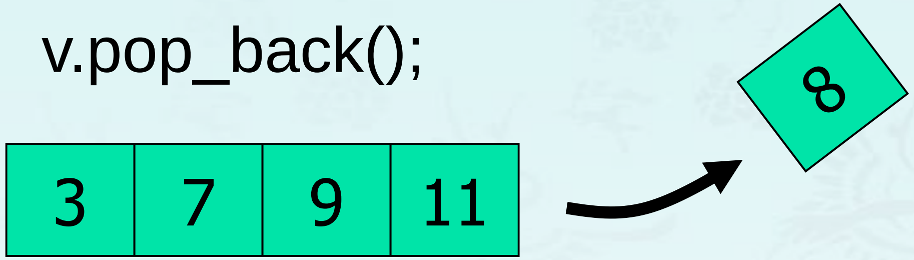
Useful if arr is an argument in a function

Vector Container

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

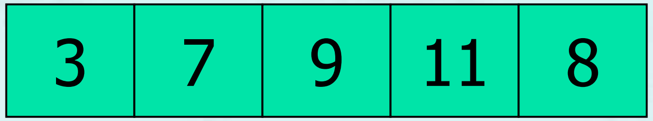
3	7	9	11	8

v.pop_back();



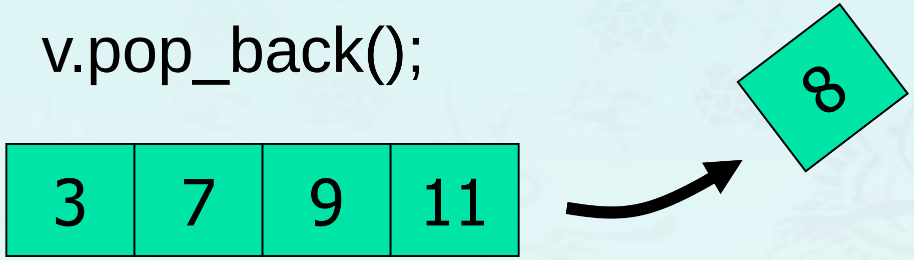
Vector Container

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



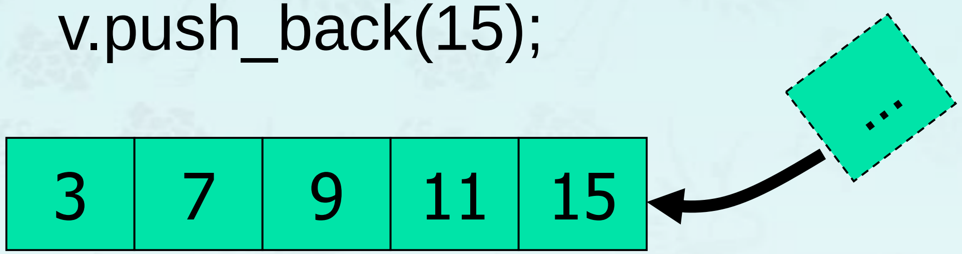
3	7	9	11	8

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



3	7	9	11

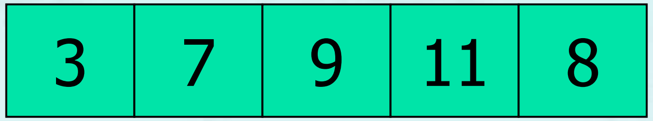
```
v.push_back(15);
```



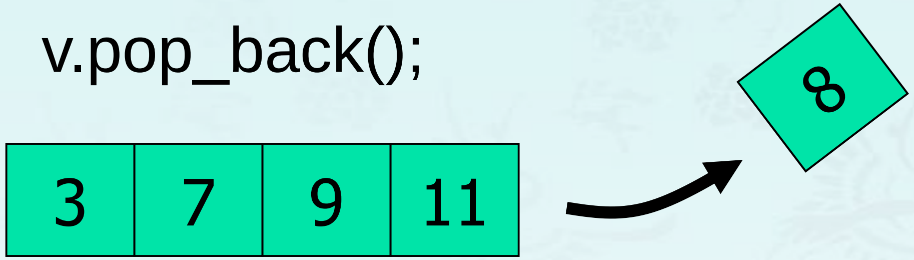
3	7	9	11	15

Vector Container

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

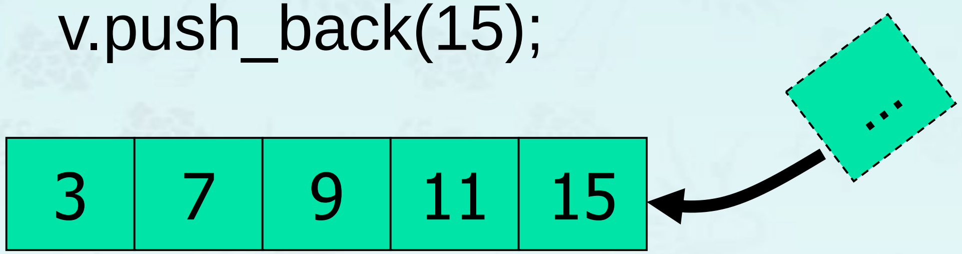


3	7	9	11	8



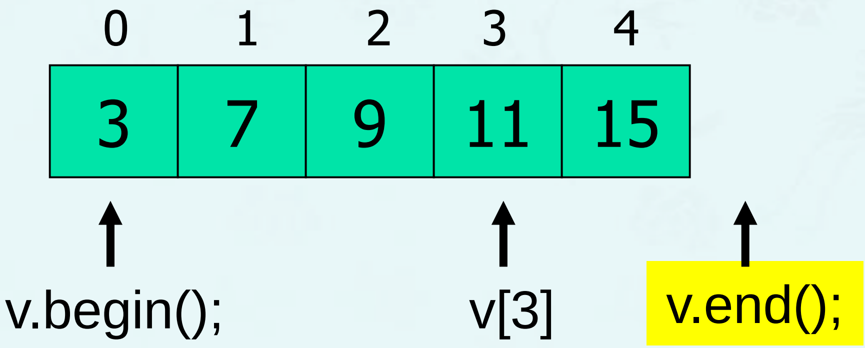
v.pop_back();

3	7	9	11



v.push_back(15);

3	7	9	11	15



0	1	2	3	4
3	7	9	11	15

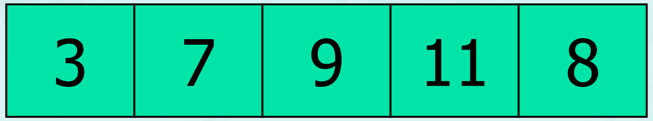
v.begin();

v[3]

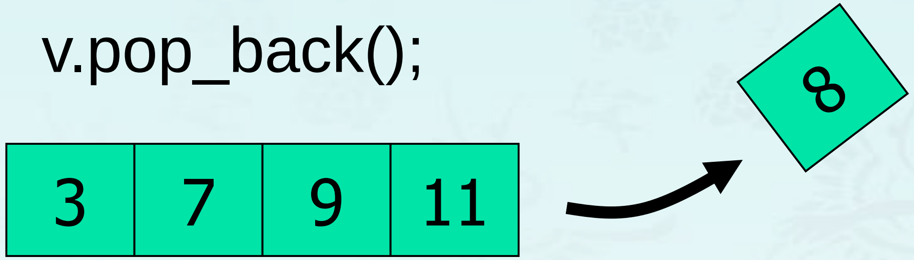
v.end();

Vector Container

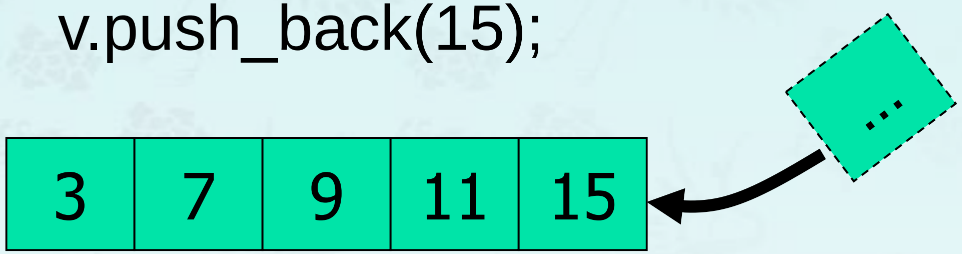
```
vector<int> v{3, 7, 9, 11, 8};
```

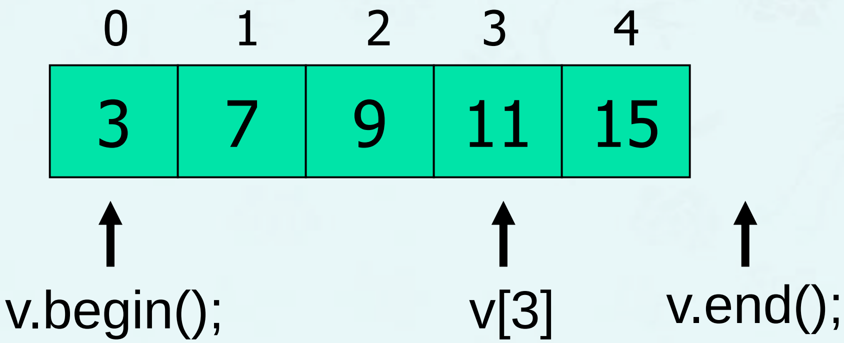


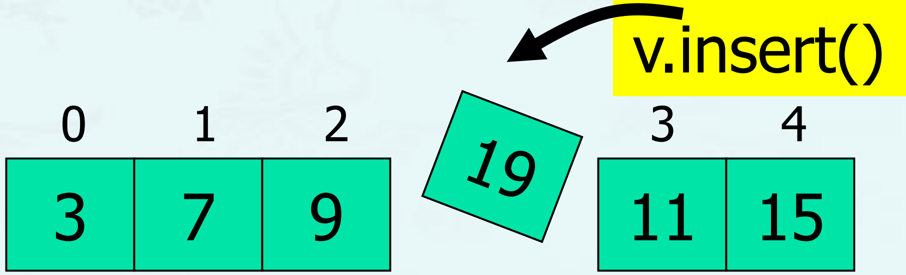
v.pop_back();



v.push_back(15);

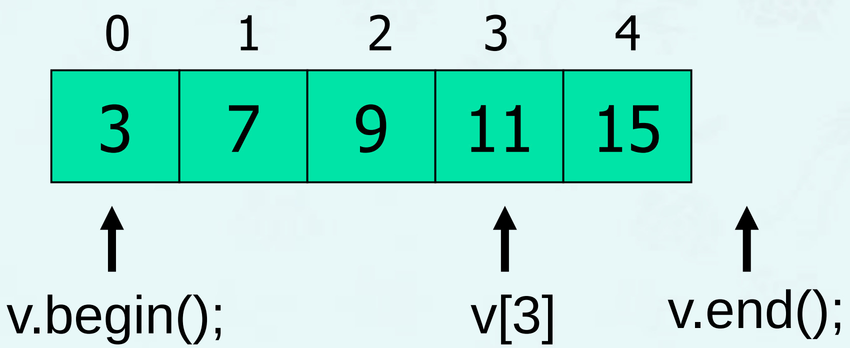


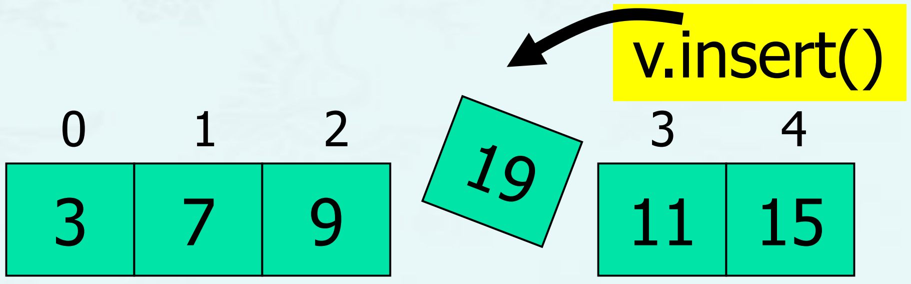




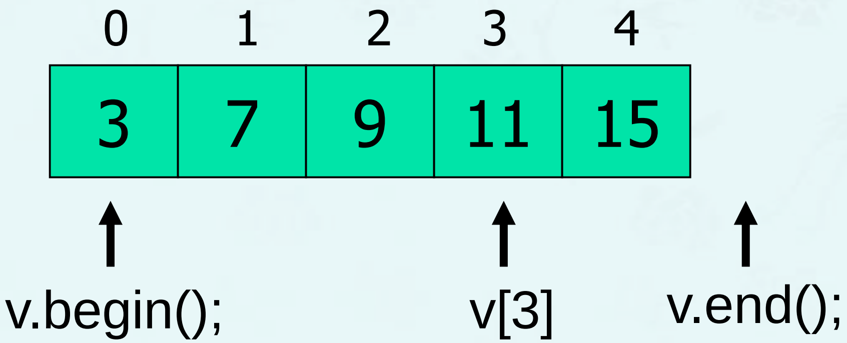
Vector Container – insert()

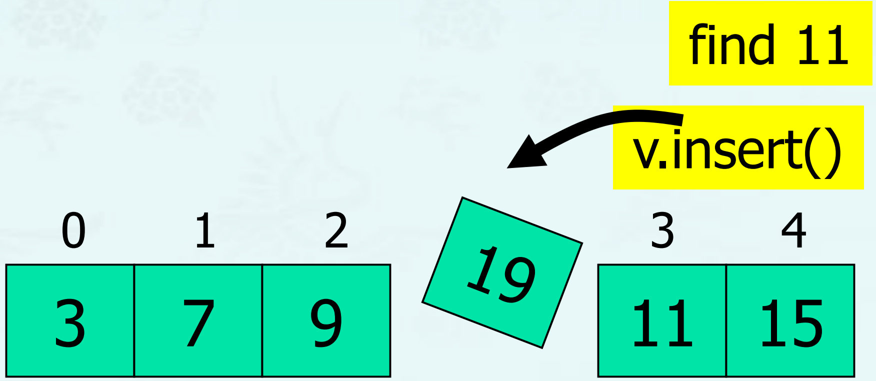
```
int main() {
  vector<int> vec{ 3, 7, 9, 11, 15 };

  vec.insert( vec.begin() + 3, 19 );
  for( auto x: vec )
    cout << x << " ";
  cout << endl;
}
```





Vector Container – find()

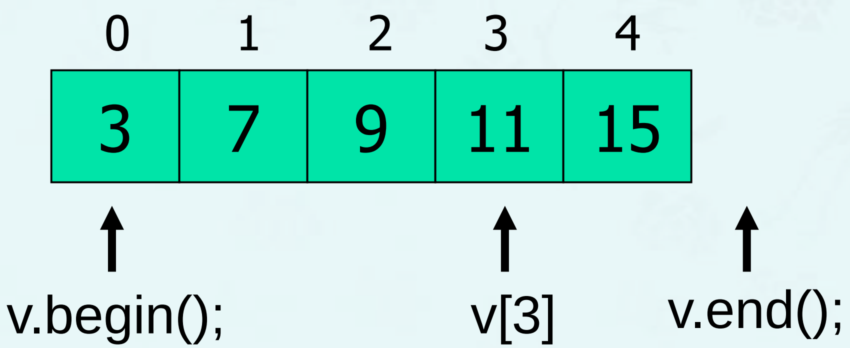


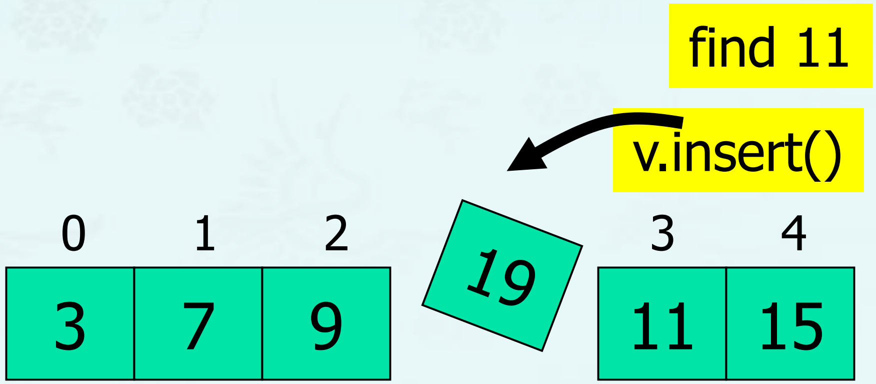


Vector Container – find()

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

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

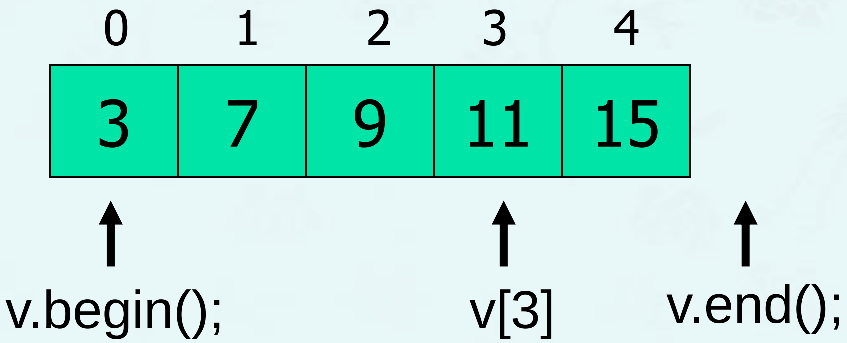
auto

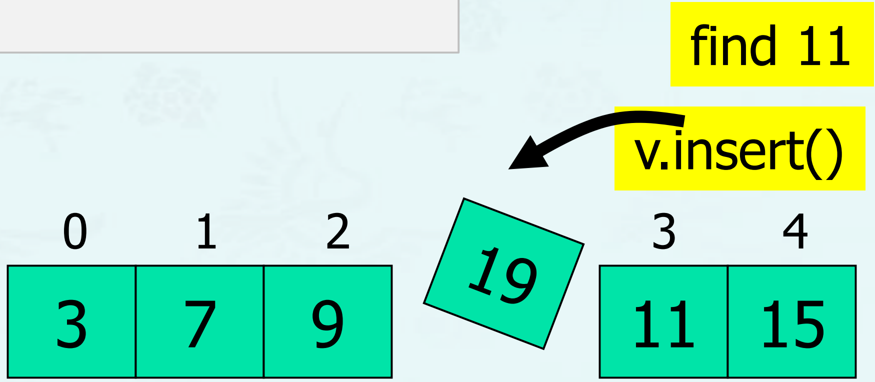




Vector Container – find()

```
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 << " ";
}
```

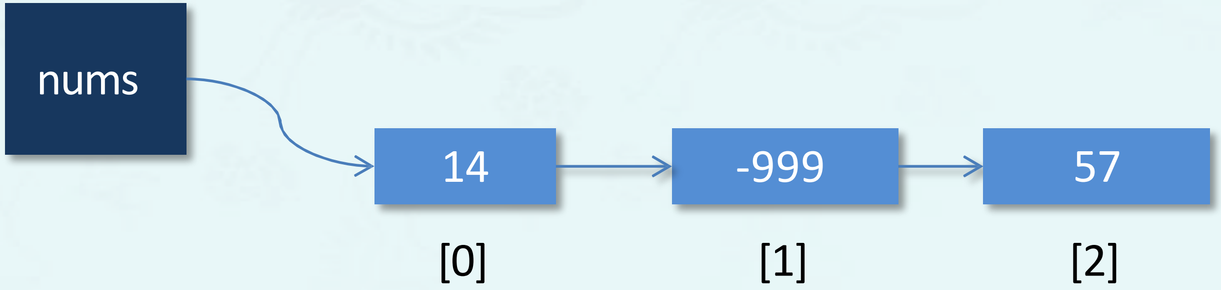




Vector Container Example

```
#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));

                              auto
  vector<Member>::iterator 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 it = 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

```cpp
#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;
}
```

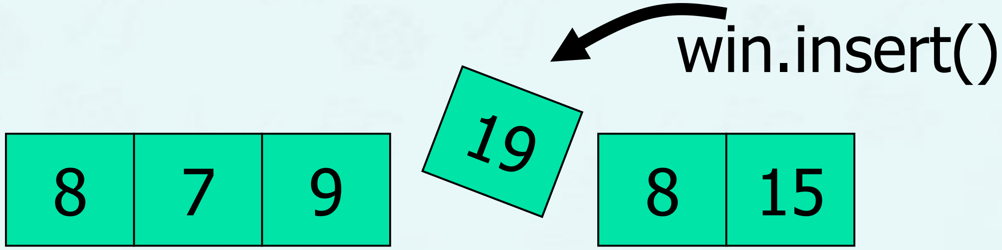
List Container

- 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

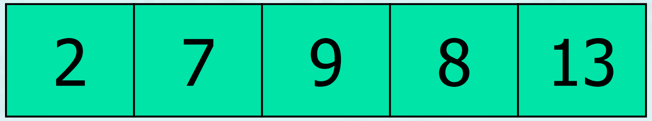
```
int arr[] = {2, 7, 9, 8, 13 };
list<int> win(arr, arr+5);
```

2	7	9	8	13

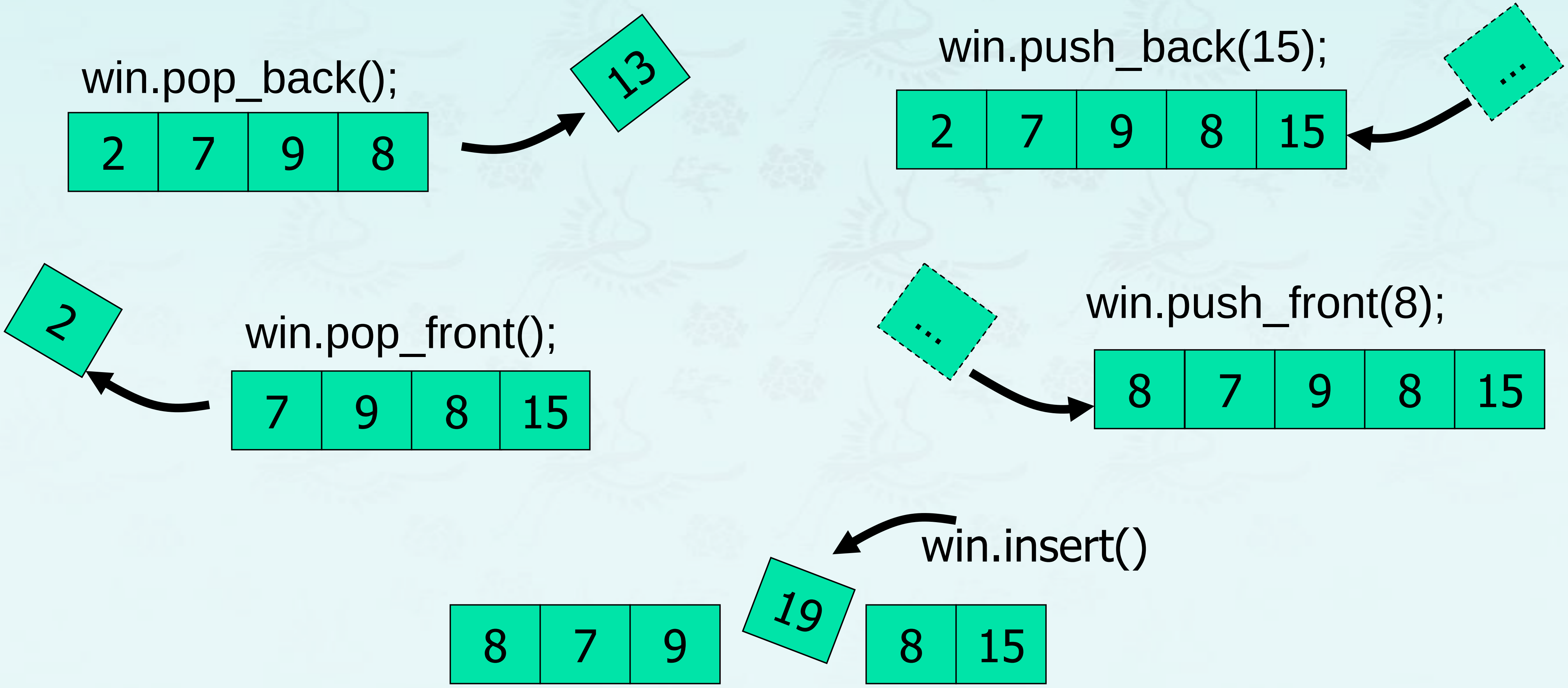


List Container

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



2	7	9	8	13



win.pop_back();

2	7	9	8

win.push_back(15);

2	7	9	8	15

win.pop_front();

7	9	8	15

win.push_front(8);

8	7	9	8	15

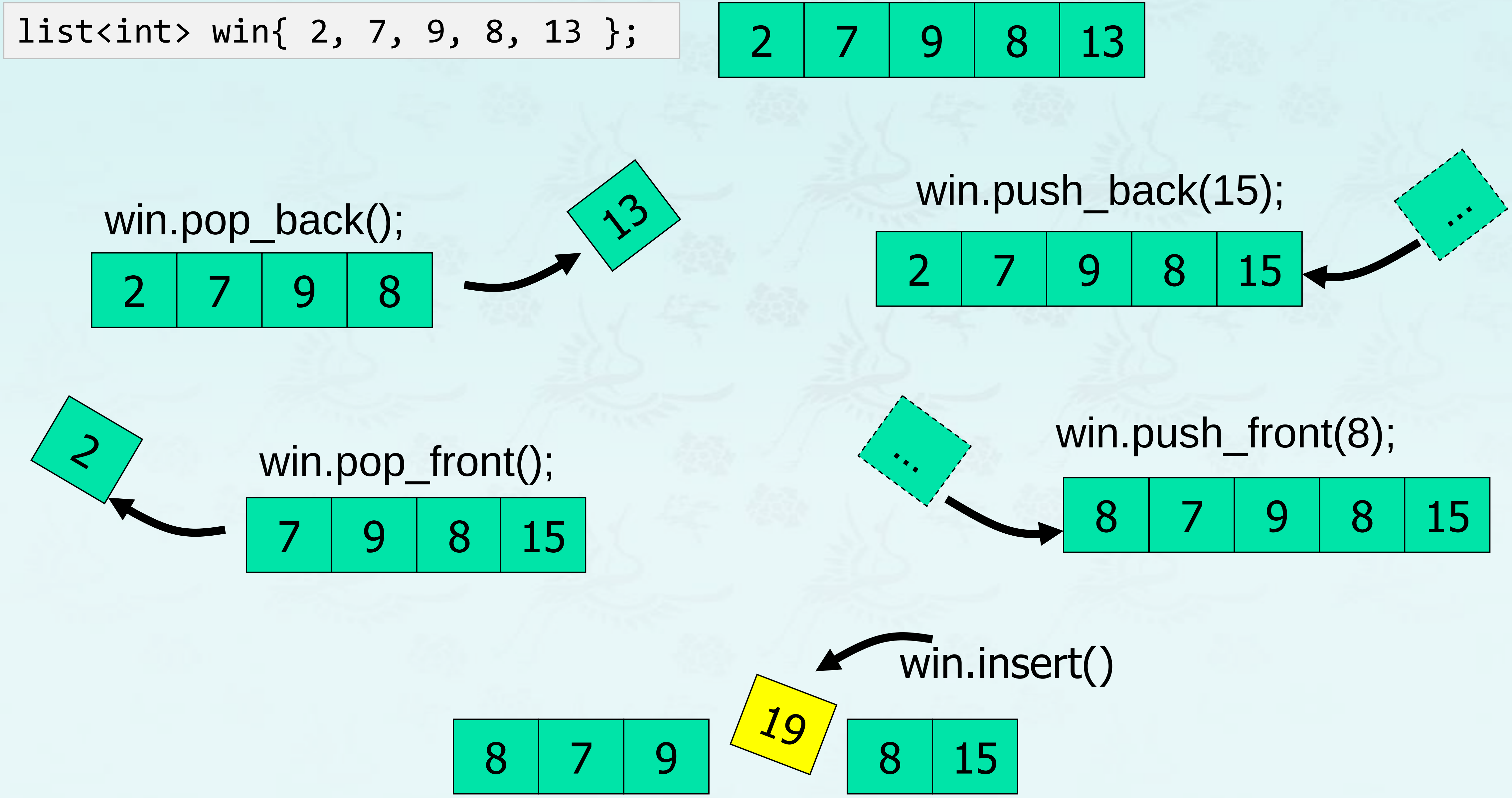
win.insert()

8	7	9

8	15

List Container

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



win.pop_back();

win.push_back(15);

win.pop_front();

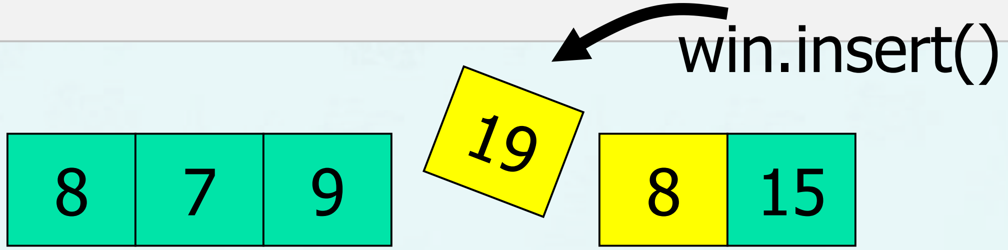
win.push_front(8);

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;
}
```



Associative Containers

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

Summary

&