Modern C++ Programming

8. Containers, Iterators, and Algorithms

Federico Busato

University of Verona, Dept. of Computer Science

2018, v1.0

Agenda

- **Containers and Iterators**
- **Sequence Containers**
  - `std::array`
  - `std::vector`
  - `std::deque`
  - `std::list`
  - `std::forward_list`
  - Operations and complexity
- **Associative Containers**
  - `std::set`, `std::map`, etc.
  - Operations and complexity
- **Container Adaptors**
  - Methods
- **Implement a Custom Iterator**
  - Iterator semantic
  - Implementation example
- **Iterator Utility Methods**
  - Iterator operations
  - Range access methods
  - Iterator traits
- **Algorithms Library**
  - Implementation example
- **Lambda Expressions**
  - Capture list
  - Capture list and classes
  - `mutable`

Containers and Iterators

Containers and Iterators

Container

A **container** is a class, a data structure, or an abstract data type, whose instances are collections of other objects

Containers store objects in an organized way that follows specific access rules

Iterator

An **iterator** is an object allowing a programmer to traverse a container

- **C++ Standard Template Library (STL)** is strongly based on containers and iterators

Iterator Concept

Iterator support a subset of the following operations:

Operation	Example
Read	`*it`
Write	`*it =`
Increment	`it++`
Decrement	`it--`
Comparison	`it1 < it2`
Random access	`it + 4`, `it[2]`

- Iterators are a generalization of pointers
- Pointers support all iterator operations

STL containers provide the following methods to get iterator objects:

- `begin()` returns an iterator pointing to the first element
- `end()` returns an iterator pointing to the end of the container (i.e. the element after the last element)

Container (Reasons to use Standard Containers)

- STL containers eliminate redundancy, and save time avoiding to write your own code (productivity)
- STL containers are implemented correctly, and they do not need to spend time to debug (reliability)
- STL containers are well-implemented and fast
- STL containers do not require external libraries
- STL containers share common interfaces, making it simple to utilize different containers without looking up member function definitions
- STL containers are well-documented and easily understood by other developers, improving the understandability and maintainability
- STL containers are thread safe. Sharing objects across threads preserve the consistency of the container

Container (Properties)

C++ Standard Template Library (STL) Containers have the following properties:

- Default constructor
- Destructor
- Copy constructor and assignment (deep copy)
- Iterator methods `begin()`, `end()`
- Support `std::swap`
- Content-based and order equality (`==, !=`)
- Lexicographic order comparison (`>, >=, <, <=`)
- `size()`*, `empty()`, and `max_size()` methods

* except for `std::forward_list`

Sequence Containers

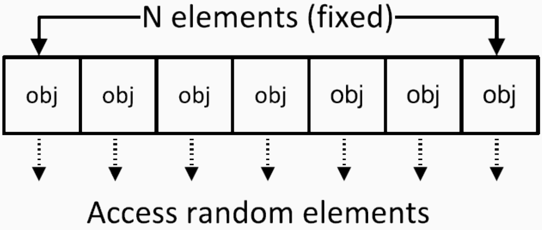
Sequence containers are data structures storing objects of the same data type in a linear mean manner

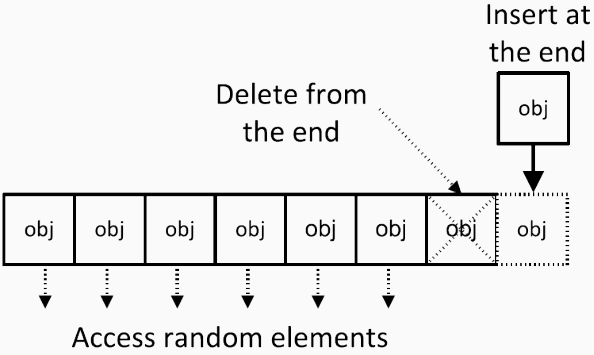
The *STL Sequence Container* types are:

- `std::array` provides a *fixed-size* contiguous array (on stack)
- `std::vector` provides a *dynamic* contiguous array
- `std::list` provides a *double-linked list*
- `std::deque` provides a *double-ended queue* (implemented as array-of-array)
- `std::forward_list` provides a *single-linked list*

While `std::string` is not included in most container lists, it actually meets the requirements of a Sequence Container

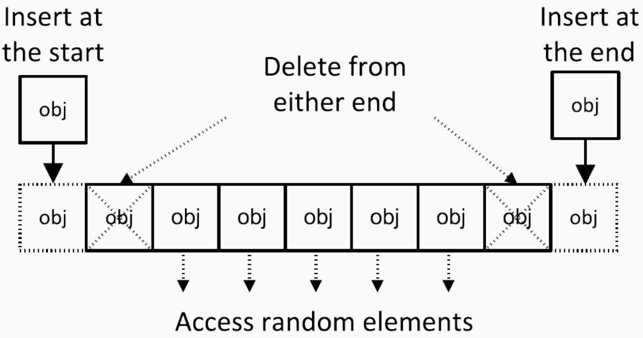
Full Story: embeddedartistry.com


N elements (fixed)
obj
obj
obj
obj
obj
obj
obj
Access random elements



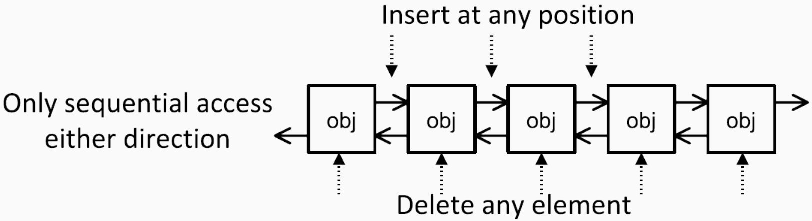
Other methods:

- **`resize()`** resizes the allocated elements of the container
- **`capacity()`** number of allocated elements
- **`reserve()`** resizes the allocated memory of the container (not size)
- **`shrink_to_fit()`** reallocate to remove unused capacity
- **`clear()`** removes all elements from the container (no reallocation)



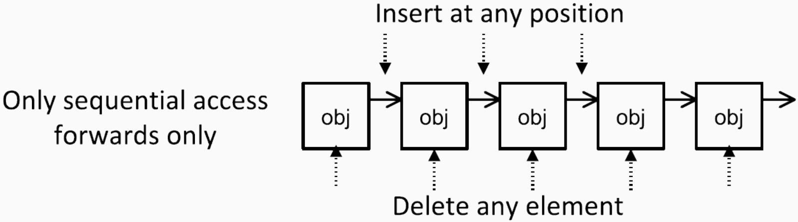
Other methods:

- `resize()` resizes the allocated elements of the container
- `shrink_to_fit()` reallocate to remove unused capacity
- `clear()` removes all elements from the container (no reallocation)



Other methods:

- **`resize()`** resizes the allocated elements of the container
- **`shrink_to_fit()`** reallocate to remove unused capacity
- **`clear()`** removes all elements from the container (no reallocation)
- **`remove()`** removes all elements satisfying specific criteria
- **`reverse()`** reverses the order of the elements
- **`unique()`** removes all consecutive duplicate elements
- **`sort()`** sorts the container elements



Other methods:

- **`resize()`** resizes the allocated elements of the container
- **`shrink_to_fit()`** reallocate to remove unused capacity
- **`clear()`** removes all elements from the container (no reallocation)
- **`remove()`** removes all elements satisfying specific criteria
- **`reverse()`** reverses the order of the elements
- **`unique()`** removes all consecutive duplicate elements
- **`sort()`** sorts the container elements

CONTAINERS	operator[]/at	front	back
std::array	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::vector	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::list		$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::deque	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::forward_list		$\mathcal{O}(1)$	

CONTAINERS	push_front	pop_front	push_back	pop_back	insert(it)	erase(it)
std::array						
std::vector			$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
std::list	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
std::deque	$\mathcal{O}(1)^*$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^* / \mathcal{O}(n)^\dagger$	$\mathcal{O}(1)$
std::forward_list	$\mathcal{O}(1)$	$\mathcal{O}(1)$			$\mathcal{O}(1)$	$\mathcal{O}(1)$

*Amortized time †Worst case (middle insertion)

Full Story: https://en.cppreference.com/w/cpp/container

```cpp
#include <array>     // <--
#include <iostream>  // std::array supports initialization
int main() {         // only throw initialization list
    std::array<int, 3> arr1 = { 5, 2, 3 };
    std::array<int, 4> arr2 = { 1, 2 };        // [3]: 0, [4]: 0
//  std::array<int, 3> arr3 = { 1, 2, 3, 4 }; // run-time error
    std::array<int, 3> arr4(arr1);             // copy constructor
    std::array<int, 3> arr5 = arr1;            // assign operator

    arr5.fill(3);                           // equal to { 3, 3, 3 }
    std::sort(arr1.begin(), arr1.end()); // arr1: 2, 3, 5
    std::cout << (arr1 > arr2);             // true

    std::cout << sizeof(arr1);              // 12
    std::cout << arr1.size();               // 3
    for (const auto& it : arr1)
        std::cout << it << ", ";            // 2, 3, 5

    std::cout << arr1[0];                   // 2
    std::cout << arr1.at(0);                // 2 (safe)
    std::cout << arr1.data()[0]             // 2 (raw array)
}
```

Sequence Containers (std::vector example)

```cpp
#include <vector>     // <--
#include <iostream>

int main() {
    std::vector<int>         vec1   { 2, 3, 4 };
    std::vector<std::string> vec2 = { "abc", "efg" };
    std::vector<int>         vec3(2);     // [0, 0]
    std::vector<int>         vec4{2};     // [2]
    std::vector<int>         vec5(5, -1); // [-1, -1, -1, -1, -1]

    vec5.fill(3);                        // equal to { 3, 3, 3 }
    std::cout << sizeof(vec1);           // 24
    std::cout << vec1.size();            // 3
    for (const auto& it : vec1)
        std::cout << it << ", ";         // 2, 3, 5

    std::cout << vec1[0];                // 2
    std::cout << vec1.at(0);             // 2 (safe)
    std::cout << vec1.data()[0]          // 2 (raw array)
    vec1.push_back(5);                   // [2, 3, 4, 5]
}
```

Sequence Containers (`std::list` example)

```cpp
#include <list>        // <--
#include <iostream>

int main() {
    std::list<int>         list1   { 2, 3, 2 };
    std::list<std::string> list2 = { "abc", "efg" };
    std::list<int>         list3(2);     // [0, 0]
    std::list<int>         list4{2};     // [2]
    std::list<int>         list5(2, -1); // [-1, -1]
    list5.fill(3);                       // [3, 3]

    list1.push_back(5);                  // [2, 3, 2, 5]
    list1.merge(arr5);                   // [2, 3, 2, 5, 3, 3]
    list1.remove(2);                     // [3, 5, 3, 3]
    list1.unique();                      // [3, 5, 3]
    list1.sort();                        // [3, 3, 5]
    list1.reverse();                     // [5, 3, 3]
}
```

Sequence Containers (`std::deque` example)

```cpp
#include <deque>        // <--
#include <iostream>

int main() {
    std::deque<int>             queue1   { 2, 3, 2 };
    std::deque<std::string> queue2 = { "abc", "efg" };
    std::deque<int>             queue3(2);      // [0, 0]
    std::deque<int>             queue4{2};      // [2]
    std::deque<int>             queue5(2, -1); // [-1, -1]
    queue5.fill(3);                             // [3, 3]

    queue1.push_front(5);                       // [5, 2, 3, 2]
    queue1[0];                                  // retuns 5
}
```

Sequence Containers (`std::forward_list` example)

```cpp
#include <forward_list>      // <--
#include <iostream>

int main() {
    std::forward_list<int>         flist1   { 2, 3, 2 };
    std::forward_list<std::string> flist2 = { "abc", "efg" };
    std::forward_list<int>         flist3(2);     // [0, 0]
    std::forward_list<int>         flist4{2};     // [2]
    std::forward_list<int>         flist5(2, -1); // [-1, -1]
    flist5.fill(4);                               // [4, 4]

    flist1.push_front(5);                       // [5, 2, 3, 2]
    flist1.insert_after(flist1.begin(), 0);     // [5, 0, 2, 3, 2]
    flist1.erase_after(flist1.begin(), 0);      // [5, 2, 3, 2]
    flist1.remove(2);                           // [3, 5, 3, 3]
    flist1.unique();                            // [3, 5, 3]
    flist1.sort();                              // [3, 3, 5]
    flist1.reverse();                           // [5, 3, 3]
    flist1.merge(flist5);                       // [5, 3, 3, 4, 4]
}
```

Associative Containers

Associative Containers (Overview)

An **associative container** is a collection of elements not necessarily indexed with sequential integers and that supports efficient retrieval of the stored elements through keys

Keys are unique

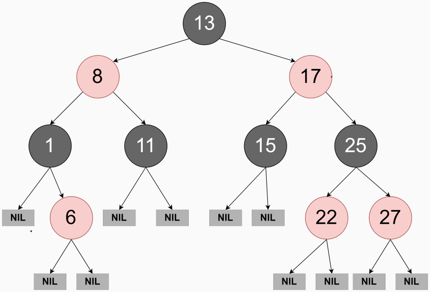
- `std::set` is a collection of sorted unique elements (`operator<`)
- `std::unsorted_set` is a collection of unsorted unique keys
- `std::map` is a collection of unique <key, value> pairs, sorted by keys
- `std::unsorted_map` is a collection of unique <key, value> pairs, unsorted

Multiple entries for the same key are permitted

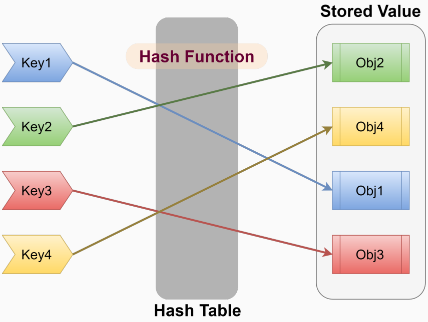
- `std::multiset` is a collection of sorted elements (`operator<`)
- `std::unsorted_multiset` is a collection of unsorted elements
- `std::multimap` is a collection of <key, value> pairs, sorted by keys
- `std::unsorted_multimap` is a collection of <key, value> pairs

Associative Containers (Internal Representation)

Note: sorted associative containers are typically implemented using *red-black trees*, while unsorted associative containers (C++11) are implemented using *hash tables*



Red-Black Tree



Hash Table

Associative Containers (Supported Operations and Complexity)

CONTAINERS	insert	erase	count	find	lower_bound upper_bound
Sorted Containers	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	
Unsorted Containers	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)^*$	$\mathcal{O}(\log(n))$

* $\mathcal{O}(n)$ worst case

- **`count()`** returns the number of elements with **`key`** equal to a specified argument
- **`find()`** returns the element with **`key`** equal to a specified argument
- **`lower_bound()`** returns an iterator pointing to the first element that is *not less* than **`key`**
- **`upper_bound()`** returns an iterator pointing to the first element that is *greater* than **`key`**

Associative Containers (Other Methods)

Sorted/Unsorted containers:

- `equal_range()` returns a range containing all elements with the given `key`

std::map, std::unsorted_map

- `operator[]/at()` returns a reference to the element having the specified key in the container. A new element is generated in the set unless the key is found

Unsorted containers:

- `bucket_count()` returns the number of buckets in the container
- `reserve()` sets the number of buckets to the number needed to accommodate at least count elements without exceeding maximum load factor and rehashes the container

Associative Containers (`std::set` example)

```cpp
#include <set>       // <--
#include <iostream>

int main() {
    std::set<int>         set1  { 5, 2, 3, 2, 7 };
    std::set<int>         set2 = { 2, 3, 2 };
    std::set<std::string> set3 = { "abc", "efg" };
    std::set<int>         set4;         // empty set

    set2.erase(2);                      // [ 3 ]
    set3.insert("hij");                 // [ "abc", "efg", "hij" ]
    for (const auto& it : set1)
        std::cout << it << " ";         // 2, 3, 5, 7  (sorted)

    auto search = set1.find(2);         // iterator
    std::cout << search != set1.end(); // true
    auto it     = set1.lower_bound(4);
    std::cout << *it;                   // 5

    set1.count(2);                      // 1, note: it can only be 0 or 1
    auto it_pair = set1.equal_range(2); // iterator between [2, 3)
}
```

Associative Containers (std::map example)

```cpp
#include <map>       // <--
#include <iostream>

int main() {
    std::map<std::string, int> map1 { {"bb", 5}, {"aa", 3} };
    std::map<double, int> map2;        // empty map

    std::cout << map1["aa"];           // prints 3
    map1["dd"] = 3;                    // insert <"dd", 3>
    map1["dd"] = 7;                    // change <"dd", 7>
    std::cout << map1["cc"];           // insert <"cc", 0>
    for (const auto& it : map1)
        std::cout << it.second << " "; // 3, 5, 0, 7

    map1.insert( {"jj", 1} );          // insert pair
    auto search = set1.find("jj");     // iterator
    std::cout << search != set1.end(); // true
    auto it     = set1.lower_bound("bb");
    std::cout << *it.second;           // 5
}
```

Associative Containers (std::multiset example)

```
#include <multiset>       // <--
#include <iostream>

int main() {
    std::multiset<int>    mset1 {1, 2, 5, 2, 2};
    std::multiset<double> mset2;   // empty map

    mset1.insert(5);
    for (const auto& it : mset1)
        std::cout << it << " ";    // 1, 2, 2, 2, 5, 5
    std::cout << mset1.count(2);   // prints 3

    auto it = mset1.find(3);       // iterator
    std::cout << *it << " " << *(it + 1); // prints 5, 5

    it      = mset1.lower_bound(4);
    std::cout << *it;                 // 5
}
```

Container Adaptors

Container Adapters (Overview)

Container adapters are interfaces for reducing the number of functionalities normally available in a container

The underlying container of a container adapters can be optionally specified in the declaration

The *STL Container Adapters* are:

- `std::stack` LIFO data structure
  default underlying container: **`std::deque`**
- `std::queue` FIFO data structure
  default underlying container: **`std::deque`**
- `std::priority_queue` (max) priority queue
  default underlying container: **`std::vector`**

Container Adapters (Methods)

`std::stack` interface for a FILO (first-in, last-out) data structure

- **`top()`** accesses the top element
- **`push()`** inserts element at the top
- **`pop()`** removes the top element

`std::queue` interface for a FIFO (first-in, first-out) data structure

- **`front()`** access the first element
- **`back()`** access the last element
- **`push()`** inserts element at the end
- **`pop()`** removes the first element

`std::priority_queue` interface for a priority queue data structure (lookup to largest element by default)

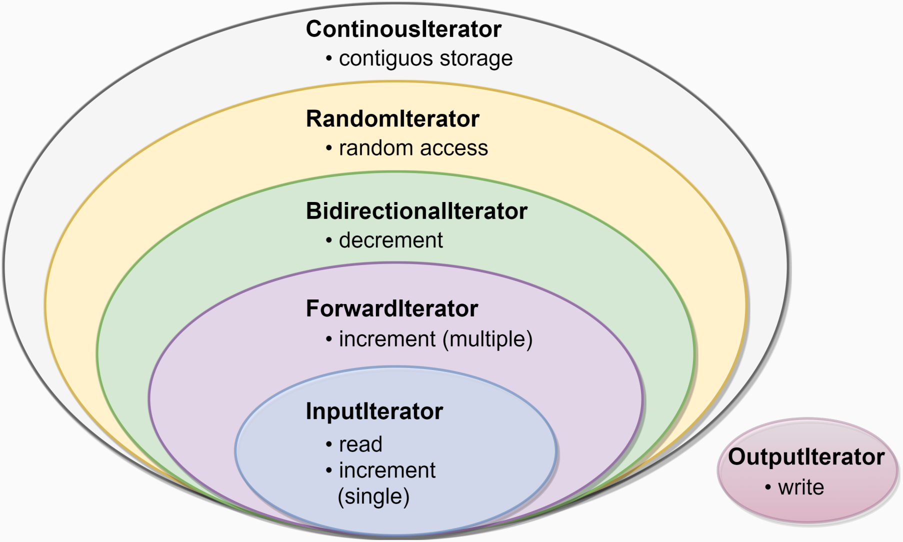
- **`top()`** accesses the top element
- **`push()`** inserts element at the end
- **`pop()`** removes the first element

Container Adaptors (examples)

```
#include <stack>          // <--
#include <queue>          // <--
#include <priority_queue> // <--
#include <iostream>

int main() {
    std::stack<int> stack1;
    stack1.push(1); stack1.push(4);   // [1, 4]
    stack1.top();     // 4
    stack1.pop();     // [1]

    std::queue<int> queue1;
    queue1.push(1); queue1.push(4);   // [1, 4]
    queue1.front();   // 1
    queue1.pop();     // [4]

    std::priority_queue<int> pqueue1;
    pqueue1.push(1); queue1.push(5); queue1.push(4);   // [5, 4, 1]
    pqueue1.top();    // 5
    pqueue1.pop();    // [4, 1]
}
```

Implement a Custom Iterator

Iterator Categories/Tags



Iterator (Iterator Semantics)

Iterator

- CopyConstructible `It(const It&)`
- CopyAssignable `It operator=(const It&)`
- Destructible `~X()`
- Dereferenceable `It_value& operator*()`
- Pre-incrementable `It& operator++()`

Input/Output Iterator

- Satisfy `Iterator`
- Equality `bool operator==(const It&)`
- Inequality `bool operator!=(const It&)`
- Post-incrementable `It operator++(int)`

Forward Iterator

- Satisfy `Input/Output Iterator`
- Default constructible `It()`
- Immutable (`const iterator`), i.e. underlying data cannot be changed

Iterator (Iterator Semantics)

Bidirectional Iterator

- Satisfy `Forward Iterator`
- Pre/post-decrementable `It& operator--(), It operator--(int)`

Random Access Iterator

- Satisfy `Bidirectional Iterator`
- Addition/Subtraction
  `void operator+(const It& it)`, `void operator+=(const It& it)`,
  `void operator-(const It& it)`, `void operator-=(const It& it)`
- Comparison
  `bool operator<(const It& it)`, `bool operator>(const It& it)`,
  `bool operator<=(const It& it)`, `bool operator>=(const It& it)`
- Subscripting `It_value& operator[](int index)`

Full Story: `anderberg.me/2016/07/04/c-custom-iterators/`

Implement a simple iterator

Goal: implement a simple iterator to iterate over `List` elements and achieve the following result:

```
#include <iostream>
// !! List implementation here
int main() {
    List list;
    list.push_back(2);
    list.push_back(4);
    list.push_back(7);

    std::cout << *std::find(list.begin(), list.end(), 4); // print 4

    for (const auto& it : list)  // range-based loop
        std::cout << it << " ";  // 2, 3, 4
}
```

Range-based loops require:

- `begin()` method
- `end()` method
- pre-increment `++it`
- not equal comparison `it != end()`
- deferencing `*it`

```cpp
struct List {
    struct Node {        // Internal Node Structure
        int    _value;   // node value
        Node* _next;     // pointer to next node
    };

    Node* head { nullptr }; // head of the list
    Node* tail { nullptr }; // tail of the list

    void push_back(int value); // insert integer value at the end

    // !! here we have to define the List iterator "It"

    It begin(); // returns an Iterator pointing to the begin of the list
    It end();   // returns an Iterator pointing to the end of the list
}
```

```
#include <iterator>     // for "std::iterator", outside List declaration

struct It : public std::iterator<std::input_iterator_tag,
                                 int> {  // int dereferencing type
    Node* _ptr;       // internal pointer
                                         // it is useful to extend
    It(Node* ptr);                       // std::iterator to inherit
                                         // common iterator fields
    int& operator*(); // deferencing

    bool operator!=(const It& it);

    It& operator++();    // pre-increment
    //---------------------------
    // no needed for std::find()
    bool operator==(const It& it); // comparison

    // no needed for std::find()
    It operator++(int);  // post-increment
};
```

```
void List::push_back(int value) {
    if (head == nullptr) {  // empty list
        head = new Node();  // head is updated
        tail = head;
    }
    else {
        tail->_next = new Node();
        tail        = tail->_next; // tail is updated
    }
    tail->_data = data;
    tail->_next = nullptr; // very important to match end() method!!
}

It List::begin() {
    return It { head };
}

It List::end() {
    return It { nullptr };   // after the last element
}
```

```
void It::It(Node* ptr) : _ptr(ptr) {}

int& It::operator*() {
    return _ptr->_data;
}

bool It::operator!=(const It& it) {
    return _ptr != it._ptr;
}

It& It::operator++() {
    _ptr = _ptr->_next;
    return *this;
}
```

Iterator Utility Methods

Iterator Operations 1/2

- `std::advance(InputIt& it, Distance n)`
  Increments a given iterator it by `n` elements
  - `InputIt` must support `input` iterator requirements
  - Modifies the iterator
  - Returns `void`
  - More general than adding a value `it + 4`
  - No performance loss if `it` satisfies `random access` iterator requirements
- `std::next(ForwardIt it, Distance n)` C++11
  Returns the `n`-th successor of the iterator
  - `ForwardIt` must support `forward` iterator requirements
  - Does not modify the iterator
  - More general than adding a value `it + 4`
  - The compiler should optimize the computation if `it` satisfies `random access` iterator requirements
  - Supports negative values if `it` satisfies `bidirectional` iterator requirements

- `std::prev(BidirectionalIt it, Distance n)` C++11
  Returns the `n`-th predecessor of the iterator
  - `InputIt` must support `bidirectional` iterator requirements
  - Does not modify the iterator
  - More general than adding a value `it + 4`
  - The compiler should optimize the computation if `it` satisfies `random` access iterator requirements

- `std::distance(InputIt start, InputIt end)`
  Returns the number of elements from `start` to `last`
  - `InputIt` must support `input` iterator requirements
  - Does not modify the iterator
  - More general than adding iterator difference `it2 - it1`
  - The compiler should optimize the computation if `it` satisfies `random` access iterator requirements
  - C++11 Supports negative values if `it` satisfies `random` iterator requirements

Iterator Operations (examples)

```cpp
#include <iterator>
#include <iostream>
#include <vector>
#include <forward_list>

int main() {
    std::vector<int> vector { 1, 2, 3 }; // random access iterator

    auto it1 = std::next(vector.begin(), 2);
    auto it2 = std::prev(vector.end(), 2);
    std::cout << *it1;   // 3
    std::cout << *it2;   // 2
    std::cout << std::distance(it2, it1); // 1

    std::advance(it2, 1);
    std::cout << *it2;   // 3

    //--------------------------------------
    std::forward_list<int> list { 1, 2, 3 }; // forward iterator
//  std::prev(list.end(), 1);               // compile error
}
```

Range Access Methods

C++11\C++14 provide a generic interface for containers, plain arrays, and `std::initializer_list` to access to the corresponding iterator. Standard method `.begin()`, `.end()` etc., are not supported by plain array and initializer list

- `std::begin` begin iterator
- `std::cbegin` begin const iterator
- `std::rbegin` begin reverse iterator
- `std::crbegin` begin const reverse iterator
- `std::end` end iterator
- `std::cend` end const iterator
- `std::rend` end reverse iterator
- `std::crend` end const reverse iterator

```
#include <iterator>
#include <iostream>

int main() {
    int array[] = { 1, 2, 3 };

    for (auto it = std::crbegin(array); it != std::crend(array); it++)
        std::cout << *it << ", ";   // 3, 2, 1
}
```

Iterator Traits

`std::iterator_traits` allows retrieving iterator properties

- `difference_type` a type that can be used to identify distance between iterators
- `value_type` the type of the values that can be obtained by dereferencing the iterator. This type is void for output iterators
- `pointer` defines a pointer to the type iterated over `value_type`
- `reference` defines a reference to the type iterated over `value_type`
- `iterator_category` the category of the iterator. Must be one of iterator category tags

Iterator Traits

```
#include <iterator>

template<typename T>
void f(const T& list) {
    using D = std::iterator_traits<T>::difference_type;   // D is std::ptrdiff_t
                                                          // (pointer difference)
                                                          // (signed size_t)
    using V = std::iterator_traits<T>::value_type;        // V is double
    using P = std::iterator_traits<T>::pointer;           // P is double*
    using R = std::iterator_traits<T>::reference;         // R is double&

    // C is BidirectionalIterator
    using C = std::iterator_traits<T>::iterator_category;
}

int main() {
    std::list<double> list;
    f(list);
}
```

Algorithms Library

C++ STL Algorithms library

The algorithms library provides functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements

- STL Algorithm library allow great flexibility which makes included functions suitable for solving real-world problem
- The user can adapt and customize the STL through the use of function objects
- Library functions work independently on containers and plain array

STL Algorithms Library (Examples)

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

struct Unary {
    bool operator()(int value) {
        return value <= 6 && value >= 3;
    }
};
struct Descending {
    bool operator()(int a, int b) {
        return a > b;
    }
};

int main() {
    std::vector<int> vector { 7, 2, 9, 4 };
    // returns an iterator pointing to the first element in the range[3, 6]
    std::find_if(vector.begin(), vector.end(), Unary());
    // sort in descending order :  { 9, 7, 4, 2 };
    std::sort(vector.begin(), vector.end(), Descending());
}
```

STL Algorithms Library (Examples) 2/2

```cpp
#include <algorithm> // it includes also std::multiplies
#include <vector>
#include <cstdlib> // std::rand

struct Unary {
    bool operator()(int value) {
        return value > 100;
    }
};

int main() {
    std::vector<int> vector { 7, 2, 9, 4 };

    int product = std::accumulate(vector.begin(), vector.end(), // product = 504
                                  1, std::multiplies<int>());
    std::srand(0);
    std::generate(vector.begin(), vector.end(), std::rand);
    // now vector has 4 random values

    std::remove_if(vector.begin(), vector.end(), Unary());
}    // remove all values > 100
```

STL Algorithms Library (Possible Implementations)

std::find

```
template<class InputIt, class T>
InputIt find(InputIt first, InputIt last, const T& value) {
    for (; first != last; ++first) {
        if (*first == value)
            return first;
    }
    return last;
}
```

std::generate

```
template<class ForwardIt, class Generator>
void generate(ForwardIt first, ForwardIt last, Generator g) {
    while (first != last)
        *first++ = g();
}
```

Lambda Expressions

Lambda Expressions (Overview)

The problem: Function objects are very verbose

Lambda Expressions (or **closure**) are **inline** local-scope function objects

Lambda expression syntax:

```
[capture clause] (parameters) { body }
```

- The brackets `[]` mark the declaration of the lambda and how the local scope arguments are captured (by-value, by-reference, etc.)
- The `parameters` of the lambda are normal function parameters (optional)
- The `body` of the lambda is a normal function body

Lambda Expressions (Examples)

```cpp
#include <algorithm>
#include <vector>
int main() {
    std::vector<int> vector { 7, 2, 9, 4 };

    // lambda is a closure object of "closure type"
    auto lambda1 = [](){ return 3; };
    int var      = lambda1();          // var = 3

    auto lambda2 = []{ return 3; };    // equivalent to lambda1
    int var      = []{ return 3; }(); // definition and evaluation

    auto lambda4 = [](int value) { return value > 5; };
    std::remove_if(vector.begin(), vector.end(), lambda4);

    // lambda expressions can be defined in the same line
    std::remove_if(vector.begin(), vector.end(),
                   [](int v) { return v > 7; });

    std::sort(vector.begin(), vector.end(),
              [](int a, int b) { return a > b; });
}
```

Lambda Expressions (Capture Lists)

Lambda expressions capture external references/variables in two ways:

- Capture by copy
- Capture by reference (can modify external variable values)

Capture list can be passed as follows

- `[]` captures nothing
- `[=]` captures all variables used in the body of the lambda **by copy**
- `[&]` captures all variables used in the body of the lambda **by reference**
- `[var1]` captured only `var1` *by copy*
- `[&var2]` captured only `var2` *by reference*
- `[var1, &var2]` captured `var1` *by copy* and `var2` *by reference*

Lambda Expressions (Capture List Examples)

```cpp
#include <algorithm>
#include <vector>
struct Unary { // equivalent to lambda2, lambda4
    int _limit;
    Unary(int limit) : _limit(limit) {}

    bool operator()(int value) const { // lambda expr. are const
        return value > _limit;
    };
};
int main() {
    std::vector<int> vector { 7, 2, 9, 4 };
    int limit = 5;

//  auto lambda1 = [](int value) { return value > limit; }; // compile error
    auto lambda2 = [=](int value) { return value > limit; };      // by value
    auto lambda3 = [&](int value) { return value > limit; };      // by ref
    auto lambda4 = [limit](int value) { return value > limit; };  // by value
    auto lambda5 = [&limit](int value) { return value > limit; }; // by ref

    std::remove_if(vector.begin(), vector.end(), lambda5);
}
```

Lambda Expressions (Capture List Other Cases)

- `[=, &var1]` captures all variables used in the body of the lambda **by copy**, except `var1` that is captured **by reference**
- `[&, var1]` captures all variables used in the body of the lambda **by reference**, except `var1` that is captured **by value**
- A lambda expression can read a variable without capturing it if the variable is a **`constexpr`**

```
int main() {
    constexpr int limit = 5;
    int var1 = 3, var2 = 4;

    auto lambda1 = [](int value) { return value > limit; };

    auto lambda2 = [=, &var3](int value) {
                        return var1 + value > var2 + var3;
                    };
}
```

Lambda Expressions (Capture List and Classes)

- `[this]` capture the current object (`*this`) by reference
- `[var = var]` capture the current object member `var` by copy C++14
- `[&var = var]` capture the current object member `var` by reference C++14

Class name conflicts:

```
class A {
    int data = 1;    // <--
    void f() {
        int var = 2; // <--
        // return 3 (nearest scope)
        auto lambda1 = [=]() { int var = 3; return var; };

        // return 2 (nearest scope)
        auto lambda2 = [=]() { return var; };

        auto lambda3 = [this]() { return this->data; };  // return 1
//      auto lambda4 = [data]() { return data; }; // compile error (not visible)
        auto lambda5 = [data = data]() { return data; }; // return 1
    }
};
```

Lambda Expressions (Other Features)

C++14 Lambda expression parameters can be initialized

```
auto func1 = [](int i = 6) { return i + 4; };
```

C++14 Lambda expression parameters can automatically deduced

```
auto func1 = [](auto value) { return value + 4; };
```

Lambda expression parameters can composed

```
int main() {
    auto lambda1 = [](int value) { return value + 4; };
    auto lambda2 = [](int value) { return value * 2; };

    auto lambda3 = [&](int value) { return lambda2(lambda1(value)); };
    // returns (value + 4) * 2
}
```

Lambda Expressions (`mutable`)

`mutable` specifier allows the lambda to modify the parameters captured by copy

```
#include <iostream>

int main() {
    int var1 = 1;
    auto lambda1 = [&]() { var1 = 4; };
    std::cout << lambda1() << ", " << var1;  // print 4, 4

    int var2 = 1;
//  auto lambda2 = [=]() { var2 = 4; };  // compile error
                                         // lambda operator() is const
    auto lambda3 = [=]() mutable { var2 = 4; };
    std::cout << lambda3() << ", " << var2;  // print 1, 4
}
```

Lambda Expressions (Other Examples)

Generate 100 numbers in the range [0, 100)

```
#include <algorithm>
#include <iostream>
#include <random>
#include <vector>

int main() {
    // Specify the seed
    auto seed = chrono::high_resolution_clock::now()
                .time_since_epoch().count();
    // Specify the engine
    std::mt19937 generator(rnd_device());
    // Specify the distribution
    std::uniform_int_distribution<int> dist(0, 100);

    std::vector<int> vector(100);
    std::generate(vector.begin(), vector.end(),
                  [&](){ return distribution(generator) } );
    std::for_each(vector.begin(), vector.end(),
                  [](auto v) { std::cout << v << " "; });
}
```