Modern C++ Programming

20. Containers, Iterators, Ranges, and Algorithms

Federico Busato

Table of Contents

Table of Contents

Table of Contents

Table of Contents

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 following specific access rules

Iterator

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

- *Iterators* are a generalization of pointers
- A pointer is the simplest *iterator*, and it supports all its operations

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

Reasons to use Standard Containers

- STL containers eliminate redundancy, and save time avoiding writing 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`

Iterator Concept

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)

There are different categories of **iterators** and each of them supports 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]`

ContinousIterator
• contiguos storage

RandomIterator
• random access

BidirectionalIterator
• decrement

ForwardIterator
• increment (multiple)

InputIterator
• read
• increment (single)

OutputIterator
• write
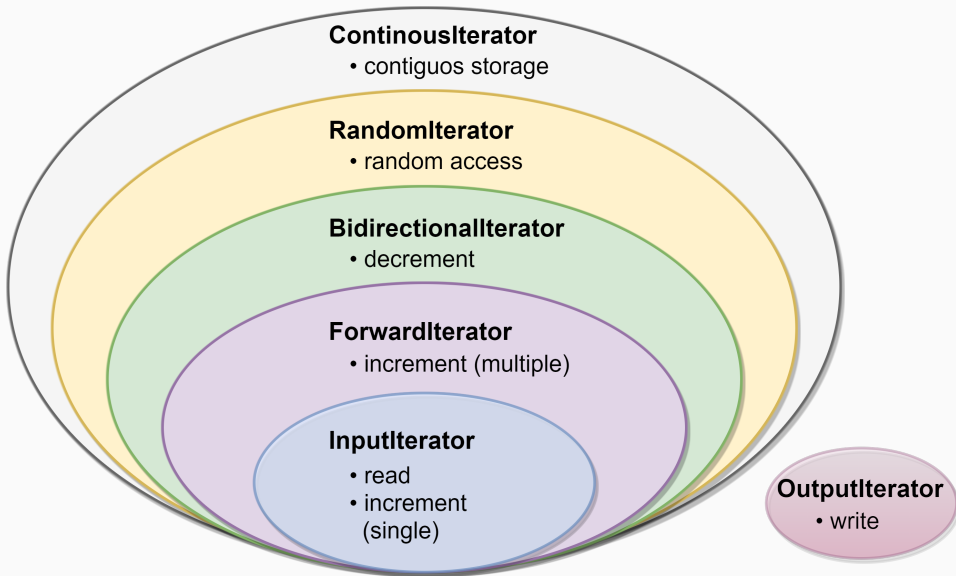
Iterator
- Copy Constructible `It(const It&)`
- Copy Assignable `It operator=(const It&)`
- Destructible `~X()`
- Dereferenceable `It_value& operator*()`
- Pre-incrementable `It& operator++()`

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()`

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)`

anderberg.me/2016/07/04/c-custom-iterators/

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 (`constexpr` in C++20)

- `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

N elements (fixed)
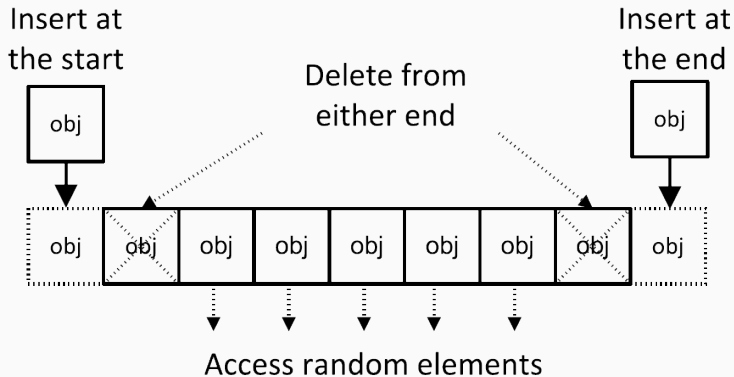
Access random elements
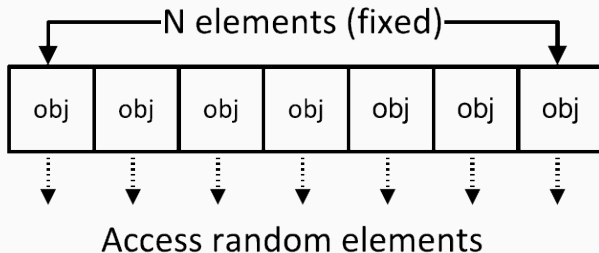
Using `std::array` instead of *raw arrays*

- ☑ *Copy semantic*, e.g. return value of a function, stored in container, etc.
- ☑ *Do not decay to a pointer*, prevent function overloading bugs
- ☑ *Out-of-bound checks* in debug mode if provided by the standard library
- ☑ *Allow zero-size arrays*
- ⚠ *Increase compile-time/binary size*

Insert at
the end

obj

Delete from
the end

obj | obj | obj | obj | obj | obj | obj | obj

Access random elements
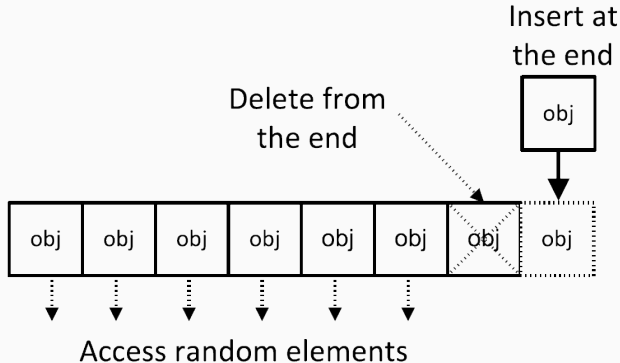
Other methods:

- `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)

`std::deque`

Insert at the start

Insert at the end

Delete from either end

obj

obj

| obj | obj | obj | obj | obj | obj | obj | obj | obj |

Access random 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)

`std::list`

Insert at any position

Only sequential access
either direction

obj → obj → obj → obj → obj →

Delete any element

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

`std::forward_list`

Insert at any position

Only sequential access
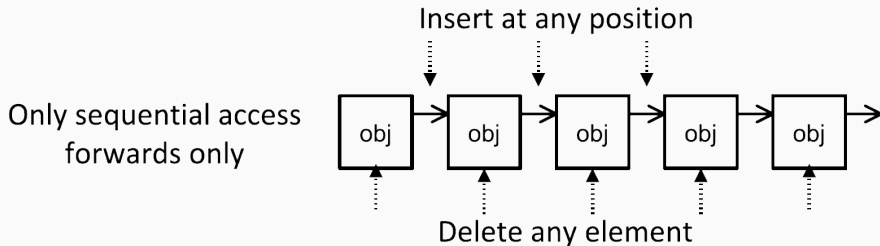forwards only

Delete any element

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

Supported Operations and Complexity

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)

std::array example

```cpp
#include <algorithm> // std::sort
#include <array>
// std::array supports initialization only through 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 }; // compiler 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
cout << (arr1 >= arr5);              // true

cout << sizeof(arr1);        // 12
cout << arr1.size();         // 3
for (const auto& it : arr1)
    cout << it << ", ";      // 2, 3, 5
cout << arr1[0];             // 2
cout << arr1.at(0);          // 2, throw if the index is not within the range
cout << arr1.data()[0];      // 2 (raw array)
```

std::vector example

```cpp
#include <vector>
#include <algorithm> // std::fill

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]

std::fill(vec5.begin(), vec5.end(), 3); // equal to { 3, 3, 3, 3, 3 }
cout << sizeof(vec1);            // 24
cout << vec1.size();            // 3
for (const auto& it : vec1)
    cout << it << ", ";          // 2, 3, 4

cout << vec1[0];        // 2
cout << vec1.at(0);     // 2 (bound check)
cout << vec1.data()[0]  // 2 (raw array)
vec1.push_back(5);      // [2, 3, 4, 5]
```

std::list example

```cpp
#include <list>
#include <algorithm> // std::fill

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]
std::fill(list5.begin(), list5.end(), 3); // [3, 3]

list1.push_back(5);                       // [2, 3, 2, 5]
list1.sort();                             // [2, 2, 3, 5]
list1.merge(list5);                       // [2, 2, 3, 3, 3, 5] merge two sorted lists
list1.remove(2);                          // [-1, -1, 3, 5]
list1.unique();                           // [-1, 3, 5]
list1.reverse();                          // [5, 3, -1]
```

std::deque example

```cpp
#include <deque>
#include <algorithm> // std::fill

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]
std::fill(queue5.begin(), queue5.end(), 3); // [3, 3]

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

std::forward_list example

```cpp
#include <forward_list>
#include <algorithm> // std::fill

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]
std::fill(flist5.begin(), flist5.end(), 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());    // [5, 2, 3, 2]
flist1.remove(2);                      // [5, 3, 3]
flist1.unique();                       // [5, 3]
flist1.reverse();                      // [3, 5]
flist1.sort();                         // [3, 5]
flist1.merge(flist5);                  // [3, 4, 4, 5] merge two sorted lists
```

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::unordered_set` is a collection of unsorted unique keys
- `std::map` is a collection of unique <key, value> pairs, sorted by keys
- `std::unordered_map` is a collection of unique <key, value> pairs, unsorted

Multiple entries for the same key are permitted

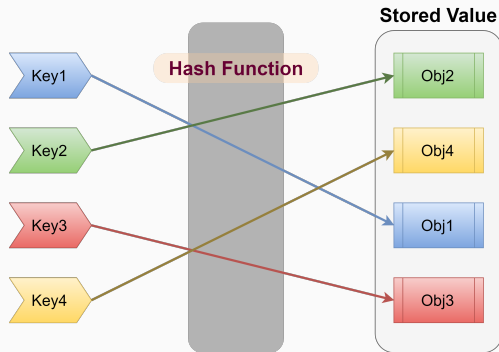- `std::multiset` is a collection of sorted elements (`operator<`)
- `std::unordered_multiset` is a collection of unsorted elements
- `std::multimap` is a collection of <key, value> pairs, sorted by keys

Sorted associative containers are typically implemented using *red-black trees*, while unordered associative containers (C++11) are implemented using *hash tables*
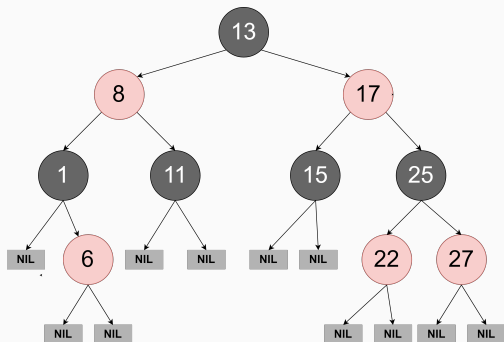
Red-Black Tree

Hash Table

Supported Operations and Complexity

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

* $\mathcal{O}(n)$ 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`

Other Methods

Ordered/Unordered containers:

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

std::map, std::unordered_map

- `operator[]/at()` returns a reference to the element having the specified key in the container.
- `operator[]` if the key is not found, it returns a new element
- `at()` if the key is not found, raises an exception

Unordered 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

std::set example

```cpp
#include <set>

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)
    cout << it << " ";               // 2, 3, 5, 7 (sorted)

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

std::map example

```cpp
#include <map>

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

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

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

std::multiset example

```cpp
#include <set> // std::multiset

std::multiset<int>    mset1 {1, 2, 5, 2, 2}; // 1, 2, 2, 2, 5
std::multiset<double> mset2;    // empty set

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

auto it = mset1.find(5);        // iterator
cout << *it;                    // 5

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

Container Adaptors

Overview

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

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

The *STL Container Adaptors* 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 Adaptors 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 the largest element by default)
- **top()** accesses the top element
- **push()** inserts an element on the proper, sorted position
- **pop()** removes the first/top element

Container Adaptor Examples

```cpp
#include <stack>          // <--
#include <queue>          // <-- also include priority_queue

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); pqueue1.push(5); pqueue1.push(4);   // [5, 4, 1]
pqueue1.top();    // 5
pqueue1.pop();    // [4, 1]
```

Implement a Custom Iterator

Goal: implement a simple iterator to iterate over a `List` of elements:

```cpp
#include <iostream>
#include <algorithm>
// !! 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, 4, 7
}
```

Range-based loops require: `begin()`, `end()`, pre-increment `++it`, not equal comparison
`it != end()`, dereferencing `*it`

```cpp
using value_t = int;

struct List {
    struct Node {              // Internal Node Structure
        value_t _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(const value_t& value); // insert a value at the end

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

    It begin() { return It{_head};   }   // begin of the list
    It end()   { return It{nullptr}; }   // end of the list
};
```

```cpp
void List::push_back(const value_t& value) {
    auto new_node = new Node{value, nullptr};
    if (_head == nullptr) { // empty list
        _head = new_node; // head is updated
        _tail = _head;
        return;
    }
    assert(_tail != nullptr);
    _tail->_next = new_node; // add new node at the end
    _tail        = new_node; // tail is updated
}
```

```cpp
struct It {
    Node* _ptr;          // internal pointer

    It(Node* ptr);       // Constructor

    value_t& operator*(); // Dereferencing

    // Not equal -> stop traversing
    friend bool operator!=(const It& itA, const It& itB);

    It& operator++();     // Pre-increment

    It operator++(int);   // Post-increment

// !! Type traits here
};
```

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

value_t& Lis::It::operator*() { return _ptr->_value; }

bool operator!=(const It& itA, const It& itB) {
    return itA._ptr != itB._ptr;
}

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

The *type traits* of an iterator describe its properties, e.g. the type of the value held, and they are widely used in the `std` algorithms

`std::iterator` class template defines the type traits for an iterator. It has been deprecated in C++17, so users need to provide the type traits explicitly

```cpp
#include <iterator>

// !! Type traits
using iterator_category = std::forward_iterator_tag;
using difference_type   = std::ptrdiff_t;
using value_type        = value_t;
using pointer           = value_t*;
using reference         = value_t&;
```

---

internalpointers.com/post/writing-custom-iterators-modern-cpp
Preparation for std::iterator Being Deprecated

Iterator Notes

Common Errors

Modify a container with a "active" iterators

```cpp
#include <vector>

std::vector<int> vec{1, 2, 3, 4, 5};
for (auto x : vec)
    vec.push_back(x);   // iterator invalidation!!
```

Iterator Utility Methods

- `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

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
}
```

Container Access Methods

C++11 provides a generic interface for <u>containers</u>, <u>plain arrays</u>, and <u>std::initializer_list</u> 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

```cpp
#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
```

`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

```cpp
#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

STL Algorithms Library

C++ STL Algorithms library

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

- STL Algorithm library allow <u>great flexibility</u> which makes included functions suitable for solving real-world problem

- The user can adapt and customize the STL through the use of <u>function objects</u>

- Library functions work independently on containers and plain array

- Many of them support `constexpr` in C++20

```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());
}
```

```cpp
#include <algorithm> // it includes also std::multiplies
#include <vector>
#include <cstdlib>   // std::rand
#include <numeric>   // std::accumulate
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::generate(vector.begin(), vector.end(), std::rand);
    // now vector has 4 random values

    // remove all values > 100 using Erase-remove idiom
    auto new_end = std::remove_if(vector.begin(), vector.end(), Unary());
    // elements are removed, but vector size is still unchanged
    vector.erase(new_end, vector.end()); // shrink vector to finish removal
}
```

50/69

STL Algorithms Library (Possible Implementations)

`std::find`

```cpp
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`

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

- `swap(v1, v2)` Swaps the values of two objects

- `min(x, y)` Finds the minimum value between x and y

- `max(x, y)` Finds the maximum value between x and y

- `min_element(begin, end)` (returns a pointer)
  Finds the minimum element in the range [begin, end)

- `max_element(begin, end)` (returns a pointer)
  Finds the maximum element in the range [begin, end)

- `minmax_element(begin, end)` C++11 (returns pointers <min,max>)
  Finds the minimum and the maximum element in the range [begin, end)

en.cppreference.com/w/cpp/algorithm

- `equal(begin1, end1, begin2)`
  Determines if two sequences are the same in
  [begin1, end1), [begin2, begin2 + end1 - begin1)

- `mismatch(begin1, end1, begin2)` (returns pointers <pos1,pos2>)
  Finds the first position where two ranges differ in
  [begin1, end1), [begin2, begin2 + end1 - begin1)

- `find(begin, end, value)` (returns a pointer)
  Finds the first element in the range [begin, end) equal to value

- `count(begin, end, value)`
  Counts the number of elements in the range [begin, end) equal to value

- `sort`(begin, end) (in-place)
  Sorts the elements in the range [begin, end) in ascending order

- `merge`(begin1, end1, begin2, end2, output)
  Merges two sorted ranges [begin1, end1), [begin2, end2), and store the results in
  [output, output + end1 - start1)

- `unique`(begin, end) (in-place)
  Removes <u>consecutive</u> duplicate elements in the range [begin, end)

- `binary search`(begin, end, value)
  Determines if an element value exists in the (sorted) range [begin, end)

- `accumulate`(begin, end, value)
  Sums up the range [begin, end) of elements with initial value (common case equal to
  zero)

- `partial_sum`(begin, end, output) (in-place)
  Computes the inclusive prefix-sum of the range [begin, end)

- `fill`(begin, end, value)
  Fills a range of elements [begin, end) with value

- `iota`(begin, end, value) C++11
  Fills the range [begin, end) with successive increments of the starting value

- `copy`(begin1, end1, begin2)
  Copies the range of elements [begin1, end1) to the new location
  [begin2, begin2 + end1 - begin1)

- `swap_ranges`(begin1, end1, begin2)
  Swaps two ranges of elements
  [begin1, end1), [begin2, begin2 + end1 - begin1)

- `remove`(begin, end, value)   (in-place)
  Removes elements equal to value in the range [begin, end)

- `includes`(begin1, end1, begin2, end2)
  Checks if the (sorted) set [begin1, end1) is a subset of [begin2, end2)

- `set_difference(begin1, end1, begin2, end2, output)`
  Computes the difference between two (sorted) sets

- `set_intersection(begin1, end1, begin2, end2, output)`
  Computes the intersection of two (sorted) sets

- `set_symmetric_difference(begin1, end1, begin2, end2, output)`
  Computes the symmetric difference between two (sorted) sets

- `set_union(begin1, end1, begin2, end2, output)`
  Computes the union of two (sorted) sets

- `make_heap(begin, end)` Creates a max heap out of the range of elements

- `push_heap(begin, end)` Adds an element to a max heap

- `pop_heap(begin, end)` Remove an element (top) to a max heap

Algorithm Library - Other Examples

```cpp
#include <algorithm>

int a        = std::max(2, 5); // a = 5
int array1[] = {7, 6, -1, 6, 3};
int array2[] = {8, 2, 0, 3, 7};

int  b = *std::max_element(array1, array1 + 5);   // b = 7
auto c = std::minmax_element(array1, array1 + 5);
//*c.first = -1, *c.second = 7
bool d = std::equal(array1, array1 + 5, array2); // d = false

std::sort(array1, array1 + 5);                    // [-1, 3, 6, 6, 7]
std::unique(array1, array1 + 5);                  // [-1, 3, 6, 7]
int e = std::accumulate(array1, array1 + 4, 0); // 15
std::partial_sum(array1, array1 + 4, array1);    // [-1, 2, 8, 15]
std::iota(array1, array1 + 5, 2);                // [2, 3, 4, 5, 6]
std::make_heap(array2, array2 + 5);              // [8, 7, 0, 3, 2]
```

C++20 Ranges

C++20 Ranges

Ranges are an abstraction that allows to operate on elements of data structures uniformly. They are an extension of the standard *iterators*

A **range** is an object that provides `begin()` and `end()` methods (an *iterator* + a *sentinel*)

`begin()` returns an *iterator*, which can be incremented until it reaches `end()`

```cpp
template<typename T>
concept range = requires(T& t) {
    ranges::begin(t);
    ranges::end(t);
};
```

- An Overview of Standard Ranges
- Range, Algorithms, Views, and Actions - A Comprehensive Guide
- Eric Nielbler - Range v3
- Range by Example

Key Concepts

Range View is a *range* defined on top of another *range*

Range Adaptors are utilities to transform a *range* into a *view*

Range Factory is a *view* that contains no elements

Range Algorithms are library-provided functions that directly operate on ranges (corresponding to std iterator algorithm)

Range Action is an object that modifies the underlying data of a range

A **range view** is a *range* defined on top of another *range* that transforms the underlying way to access internal data

- *Views* do <u>not</u> own any data

- *copy*, *move*, *assignment* operations perform in constant time

- *Views* are *composable*

- *Views* are *lazy evaluated*

Syntax:

```
range/view | view
```

```cpp
#include <iostream>
#include <ranges>
#include <vector>

std::vector<int> v{1, 2, 3, 4};

for (int x : v | std::views::reverse)
    std::cout << x << " ";  // print: "4, 3, 2, 1"

auto rv2 = v | std::views::reverse;  // cheap, it does not copy "v"

auto rv3 = v | std::views::drop(2) | // drop the first two elements
               std::views::reverse;
for (int x : rv3) // lazy evaluated
    std::cout << x << " ";  // print: "4, 3"
```

Range Adaptors are utilities to transform a *range* into a *view* with custom behaviors

- *Range adaptors* produce lazily evaluated *views*
- *Range adaptors* can be chained or composed (pipeline)

Syntax:

```
adaptor(range/view, args...)
adaptor(args...)(range/view)
range/view | adaptor(args...) // preferred syntax
```

```
#include <ranges>
#include <vector>

std::vector<int> v{1, 2, 3, 4};

for (int x : std::ranges::reverse_view(v))     // adaptor
    cout << x << " ";  // print: "4, 3, 2, 1"

auto rv2 = std::ranges::reverse_view(v);  // cheap, it does not copy "v"

auto rv3 = std::ranges::reverse_view(
            std::ranges::drop_view(2, v)); // drop the first two elements
for (int x : rv3) // lazy evaluated
    cout << x << " ";  // print: "4, 3"
```

Range Factory

Range Factory produces a *view* that contains no elements

```cpp
#include <ranges>

for (int x : std::ranges::iota_view{1, 4}) // factory (adaptor)
    cout << x << " ";                       // print: "1, 2, 3, 4"

for (int x : std::views::repeat('a', 4))   // factory (view)
    cout << x << " ";                       // print: "a, a, a, a"
```

Range Algorithms

The **range algorithms** are almost identical to the corresponding *iterator-pair* algorithms in the std namespace, except that they have *concept*-enforced constraints and accept *range* arguments

- *Range algorithms* are immediately evaluated
- *Range algorithms* can work directly on containers (`begin()` , `end()` are no more explicitly needed) and *view*

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

std::vector<int> vec{3, 2, 1};
std::ranges::sort(vec); // 1, 2, 3
```

Algorithm Operators and Projections

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

struct Data {
    char value1;
    int  value2;
};

std::vector<int> vec{4, 2, 5};
auto cmp = [](auto a, auto b) { return a > b; }; // Unary boolean predicate
std::ranges::sort(vec, cmp); // 5, 4, 2

std::vector<Data> vec2{{'a', 4}, {'b', 2}, {'c', 5}};
std::ranges::sort(vec2, {}, &Data::value2); // Projection: 2, 4, 5
                                            // {'b', 2}, {'a', 4}, {'c', 5}
```

Algorithms and Views

```cpp
// sum of the squares of the first 'count' numbers
auto sum_of_squares(int count) {
    auto squares = std::views::iota(1, count) |
                   std::views::transform([](int x) { return x * x; });
    return std::ranges::fold_left_first(squares, std::plus{});
}
```

The **range actions** mimic *std algorithms* and *range algorithms* adding the
composability property

- *Range actions* are *eager* evaluated
- *Range algorithms* work directly on *ranges*
- <u>Not</u> included in the std library

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

std::vector<int> vec{3, 5, 6, 3, 5}
// in-place
vec = vec | actions::sort    // 3, 3, 5, 5, 6
          | actions::unique;  // 3, 5, 6

vec |= actions::sort    // 3, 3, 5, 5, 6
    | actions::unique;  // 3, 5, 6
// out-of-place
auto vec2 = std::move(vec) | actions::sort    // 3, 3, 5, 5, 6
                           | actions::unique;  // 3, 5, 6
```