Филипп Хандельянц

Лекция 5/12

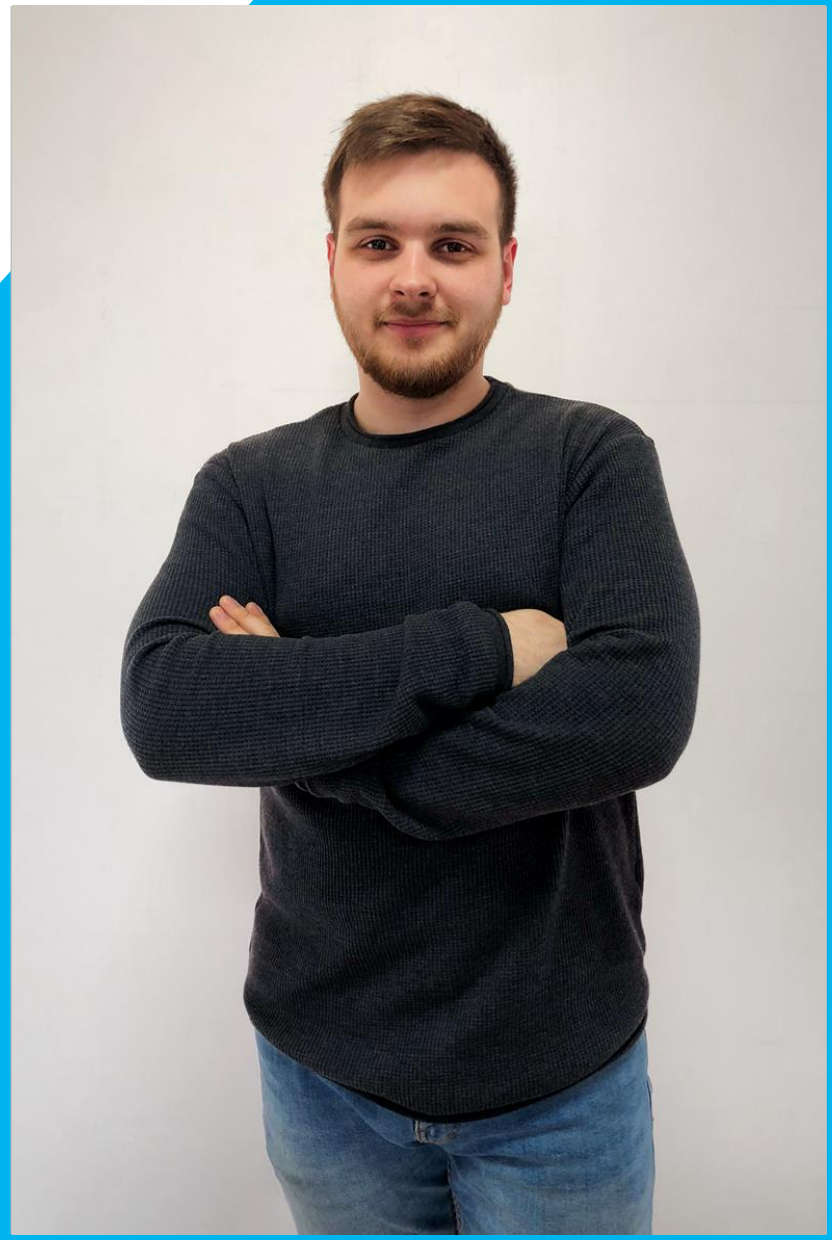
STL: концепция, контейнеры, итераторы

Докладчик

Хандельянц

Филипп Александрович

- Ведущий разработчик в команде PVS-Studio (C++/C#)
- 3 года участвую в разработке ядра C++ анализатора
- Автор статей о проверке open source-проектов



Problems

```
void foo(uint64_t count)
{
  int *arr = (int*) malloc(count * sizeof(int));
  if (arr == nullptr)
  {
    // handle memory allocation error
  }

  for (size_t i = 0; i < count; ++i)
  {
    arr[i] = 0;
  }

  // some manipulations with arr

  free(arr);
}
```

```
void foo(uint64_t count)
{
  int *arr = new int[count]{};

  // some manipulations with arr

  delete[] arr;
}
```

```
void foo(uint64_t count)
{
  int *arr = new int[count]{};

  // some manipulations with arr

  // need to increase capacity of arr
  {
    int *buf = new int[newCount];
    for (size_t i = 0; i < count; ++i)
    {
      buf[i] = arr[i];
    }

    delete[] arr;
    arr = buf;
  }

  // some manipulations with new arr

  delete[] arr;
}
```

```
class DynamicIntArray
{
  size_t size, capacity;
  int *p = NULL;

public:
  DynamicIntArray() : size(0), capacity(0), p(NULL) { }

  DynamicIntArray(size_t count, int value = int())
    : size(count), capacity(count), p(new int[count]()) { .... }

  int& operator[](size_t idx) { return p[idx]; }
  const int& operator[](size_t idx) const { return p[idx]; }

  void InsertAtEnd(int val) { .... }
  void Resize(size_t count) { .... }

  ~DynamicIntArray() { delete[] p; }
};
```

```
void foo(uint64_t count)
{
  DynamicIntArray arr(count);

  // some manipulations with arr

  // need to increase arr
  arr.Resize(newCount);

  // some manipulations with arr
}
```

```
void foo(uint64_t count)
{
  DynamicDoubleArray arr(count); // compile-time error

  // some manipulations with arr

  // need to increase arr
  arr.Resize(newCount);

  // some manipulations with arr
}
```

```
class DynamicDoubleArray
{
  size_t size, capacity;
  double *p = NULL;

public:
  DynamicDoubleArray() : size(0), capacity(0), p(NULL) { }

  DynamicDoubleArray(size_t count, int value = int())
    : size(count), capacity(count), p(new int[count]()) { .... }

  double& operator[](size_t idx) { return p[idx]; }
  const double& operator[](size_t idx) const { return p[idx]; }

  void InsertAtEnd(double val) { .... }
  void Resize(size_t count) { .... }

  ~DynamicIntArray() { delete[] p; }
};
```

```
#define DEFINE_DYNAMIC_ARRAY(Type, Name) \
  class Dynamic ## Name ## Array \
  { \
    size_t size, capacity; \
    Type *p = NULL; \
    \
  public: \
    Dynamic ## Name ## Array() : size(0), capacity(0), p(NULL) { } \
    \
    Dynamic ## Name ## Array(size_t count, Type value = Type()) \
      : size(count), capacity(count), p(new Type[count]()) { .... } \
    \
    Type& operator[](size_t idx) { return p[idx]; } \
    const Type& operator[](size_t idx) const { return p[idx]; } \
    \
    void InsertAtEnd(Type val) { .... } \
    void Resize(size_t count) { .... } \
    \
    ~Dynamic ## Name ## Array() { delete[] p; } \
  };
```

```
DEFINE_DYNAMIC_ARRAY(int, Int)
DEFINE_DYNAMIC_ARRAY(double, Double)

void foo(uint64_t count)
{
  DynamicIntArray int_arr(count);        // ok
  DynamicDoubleArray double_arr(count); // ok

  // some manipulations with arr

  // need to increase arr
  arr.Resize(newCount);

  // some manipulations with arr
}
```

```
class DynamicIntArray { size_t size, capacity; int *p = 0; public:
DynamicIntArray() : size(0), capacity(0), p(0) { } DynamicIntArray(size_t
count, int value = int()) : size(count), capacity(count), p(new
int[count]()) { } int& operator[](size_t idx) { return p[idx]; } const
int& operator[](size_t idx) const { return p[idx]; } void InsertAtEnd(int
val) { } void Resize(size_t count) { } ~DynamicIntArray() { delete[] p; }
};

class DynamicDoubleArray { size_t size, capacity; double *p = 0; public:
DynamicDoubleArray() : size(0), capacity(0), p(0) { }
DynamicDoubleArray(size_t count, double value = double()) : size(count),
capacity(count), p(new double[count]()) { } double& operator[](size_t
idx) { return p[idx]; } const double& operator[](size_t idx) const {
return p[idx]; } void InsertAtEnd(double val) { } void Resize(size_t
count) { } ~DynamicDoubleArray() { delete[] p; } };
```

```
template <class T>
class DynamicArray
{
  size_t size, capacity;
  T *p = NULL;

public:
  DynamicIntArray() : size(0), capacity(0), p(NULL) { }

  DynamicIntArray(size_t count, T value = T())
    : size(count), capacity(count), p(new int[count]()) { .... }

  T& operator[](size_t idx) { return p[idx]; }
  const T& operator[](size_t idx) const { return p[idx]; }

  void InsertAtEnd(T val) { .... }
  void Resize(size_t count) { .... }

  ~DynamicIntArray() { delete[] p; }
};
```

```
void foo(uint64_t count)
{
  DynamicArray<int> int_arr(count);        // ok
  DynamicArray<double> double_arr(count); // ok

  // some manipulations with arr

  // need to increase arr
  arr.Resize(newCount);

  // some manipulations with arr
}
```

Standard template library

Структура STL

- Контейнеры
- Итераторы
- Алгоритмы
- Адаптеры
- Функциональные объекты

Containers library

Контейнеры

- Последовательные
- Упорядоченные ассоциативные
- Неупорядоченные ассоциативные
- Адаптеры

std::allocator

```
// <memory>

template <class T>
struct allocator { .... }

template <class Alloc>
struct allocator_traits
{
  using allocator_type = Alloc;
  using value_type = Alloc::value_type;
  using pointer = Alloc::pointer;
  using const_pointer = Alloc::const_pointer;
  using difference_type = Alloc::difference_type;
  using size_type = Alloc::size_type;

  [[nodiscard]] static pointer allocate(Alloc &a, size_type n);

  static void deallocate(Alloc &a, pointer p, size_type n);

  template <class T, class ...Args>
  static void construct(Alloc &a, T *p, Args &&...args);

  template <class T>
  static void destroy(Alloc &a, T *p);
};
```

```
#include <memory>

void foo()
{
  std::allocator<int> a1;    // default allocator for ints
  int *p = a1.allocate(1);   // allocate memory for one int
  a1.construct(p, 7);        // construct the int
  std::cout << a[0] << '\n'; // prints '7'

  using prev_alloc = decltype(a1);
  std::allocator<std::allocator_traits<prev_alloc>::value_type> a2;

  a2.deallocate(p, 1); // standard allocator has no state
}
```

Sequence containers

std::vector

```
#include <vector>

template <class T, class Alloc = std::allocator<T>>
class vector
{
  using value_type = T;
  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	at, operator[]	O(1)
	data	O(1)
	front, back	O(1)
Capacity	empty	O(1)
	size, max_size, capacity	O(1)
	resize, reserve, shrink_to_fit	O(n)
Modifiers	clear, erase	O(n)
	insert, emplace, push_back	O(1)*, **O(n)**
	emplace_back, pop_back	O(1)
	swap	O(1)
Allocator	get_allocator	O(1)

```cpp
#include <vector>

void foo()
{
  // Initial capacity == 4
  std::vector<int> v = {0, 1, 2, 3}; // { 0 1 2 3 }

  // Vector reallocated, capacity != 4
  v.push_back(4); // { 0 1 2 3 4 }
  v.push_back(5); // { 0 1 2 3 4 5 }

  v.pop_back(); // { 0 1 2 3 4 }
  v.pop_back(); // { 0 1 2 3 }
  v.resize(6);  // { 0 1 2 3 0 0 }

  v.shrink_to_fit(); // vector capacity == 6
}
```

std::array

```
#include <array>

template <class T, size_t N>
struct array
{
  using value_type = T;
  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	at, operator[]	O(1)
	data	O(1)
	front, back	O(1)
Capacity	empty	O(1)
	size, max_size	O(1)
Modifiers	swap	O(n)

```cpp
#include <array>

void foo()
{
  std::array<int, 6> a1 { 0, 1, 2, 3, 4, 5 };

  std::cout << a1.front() << '\n'; // prints '0'
  std::cout << a1[1] << '\n';      // prints '1'
  std::cout << a1.back() << '\n';  // prints '5'

}
```

std::forward_list

```
#include <forward_list>

template <class T, class Alloc = std::allocator<T>>
class forward_list
{
  using value_type = T;
  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	front	O(1)
Capacity	empty	O(1)
	max_size	O(1)
	resize	O(n)
Modifiers	clear	O(n)
	erase_after	O(1) - O(n)
	insert_after	O(1) - O(n)
	push_front, emplace_after, emplace_front	O(1)
	pop_front	O(1)
	swap	O(1)
	merge	O(n)
List operations	splice_after	O(1) – O(n)
	remove, remove_if	O(n)
	reverse	O(n)
	sort	O(n log n)
	unique	O(n)
Allocator	get_allocator	O(1)

```cpp
#include <forward_list>

void foo()
{
  // Create two lists of integers
  std::forward_list<int> l1 = { 0, 2, 1, 4, 3 };
  std::forward_list<int> l2 = { 7, 9, 7, 5, 8 };

  l1.sort(); // 0 1 2 3 4
  l2.sort(); // 5 6 7 8 9

  l1.merge(l2); // l1 { 0 1 2 3 4 5 6 7 8 9 }
                // l2 { }

  l1.pop_front(); // 1 2 3 4 5 6 7 8 9
}
```

std::list

```
#include <list>

template <class T, class Alloc = std::allocator<T>>
class list
{
  using value_type = T;
  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	front, back	O(1)
Capacity	empty	O(1)
	size, max_size	O(1)
	resize	O(n)
Modifiers	clear	O(n)
	erase	O(n)
	insert	O(1) - O(n)
	push_front, push_back, emplace, emplace_front, emplace_back	O(1)
	pop_front, pop_back	O(1)
	swap	O(1)
	merge	O(n)
List operations	splice	O(1) – O(n)
	remove, remove_if	O(n)
	reverse	O(n)
	sort	O(n log n)
	unique	O(n)
Allocator	get_allocator	O(1)

```cpp
#include <list>

void foo()
{
  // Create a list containing integers
  std::list<int> l1 = { 0, 2, 1, 4, 3 };
  std::list<int> l2 = { 7, 9, 7, 5, 8 };

  l1.sort(); // { 0 1 2 3 4 }
  l2.sort(); // { 5 6 7 8 9 }

  l1.merge(l2); // l1 { 0 1 2 3 4 5 6 7 8 9 }
                // l2 { }

  l1.pop_front(); // { 1 2 3 4 5 6 7 8 9 }
  l1.pop_back();  // { 1 2 3 4 5 6 7 8 }
}
```

std::deque

```
#include <deque>

template <class T, class Alloc = std::allocator<T>>
class deque
{
  using value_type = T;
  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	at, operator[]	O(1)
	front, back	O(1)
Capacity	empty	O(1)
	size, max_size	O(1)
	resize, shrink_to_fit	O(n)
Modifiers	clear, erase	O(n)
	insert, emplace, push_front, push_back	O(1)*, **O(n)**
	emplace_front, emplace_back, pop_front, pop_back	O(1)
	swap	O(1)
Allocator	get_allocator	O(1)

```cpp
#include <deque>

void foo()
{
  // Create a deque containing integers
  std::deque<int> d { 1, 2, 3, 4, 5, 6, 7, 8 };

  d.push_front(0); // { 0 1 2 3 4 5 6 7 8 }
  d.push_back(9);  // { 0 1 2 3 4 5 6 7 8 9 }

  std::cout << d.front() << '\n'; // prints '0'
  std::cout << d.back() << '\n';  // prints '9'

  std::cout << d[4] << '\n'; // prints '4'
}
```

Ordered associative containers

std::set / std::multiset

```
#include <set> // or <multiset>

template <class Key,
          class Compare = std::less<Key>,
          class Alloc = std::allocator<T>>
class set // or multiset
{
  using key_type = Key;
  using value_type = Key;
  using key_compare = Compare;
  using value_compare = Compare;

  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	-	-
Capacity	empty	O(1)
	size, max_size	O(1)
Modifiers	clear	O(n)
	erase	O(1) – O(log n) – O(n)
	insert, emplace	O(log n)
	emplace_hint	O(1) – O(log n)
	swap	O(1)
	merge	O(n log n)
	extract	O(1) – O(log n)
Lookup	count, find, contains, equal_range	O(log n)
	lower_bound, upper_bound	O(log n)
Observers	key_comp, value_comp	O(1)
Allocator	get_allocator	O(1)

```cpp
#include <set>

void foo()
{
  std::set<int> s1 { 0, 1, 1, 2, 6, 3, 3, 4, 9, 5, 6, 7, 8, 9 };

  // s1 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

  std::set<int> s2;
  if (auto node = s1.extract(0))
  {
    s2.insert(node);

    // s1 { 1, 2, 3, 4, 5, 6, 7, 8, 9  }
    // s2 { 0 }
  }
}
```

std::map / std::multimap

```
#include <map> // or <multimap>

template <class Key, class T,
          class Compare = std::less<Key>,
          class Alloc = std::allocator<std::pair<const Key, T>>>
class map // or multimap
{
  using key_type = Key;
  using mapped_type = T;
  using value_type = std::pair<const Key, T>;
  using key_compare = Compare;
  using value_compare = Compare;

  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	at, operator[]	O(log n)
Capacity	empty	O(1)
	size, max_size	O(1)
Modifiers	clear	O(n)
	erase	O(1) – O(log n) – O(n)
	insert, insert_or_assign, emplace, try_emplace	O(log n)
	emplace_hint	O(1) – O(log n)
	swap	O(1)
	merge	O(n log n)
	extract	O(1) – O(log n)
Lookup	count, find, contains, equal_range	O(log n)
	lower_bound, upper_bound	O(log n)
Observers	key_comp, value_comp	O(1)
Allocator	get_allocator	O(1)

```cpp
#include <string>
#include <map>

void foo()
{
  std::map<std::string, size_t> dict { { "foo", 0 },
                                       { "bar", 1 },
                                       { "foobar", 2 } };

  auto val = dict["bar"]; // find key "bar" and return mapped value 1
  dict["barfoo"] = 3;     // insert new element std::pair { "barfoo", 3 }
}
```

Unordered associative containers

std::unordered_set / std::unoredered_multiset

```
#include <unordered_set> // or <multiset>

template <class Key,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Allocator = std::allocator<Key>>
class unordered_set // or unordered_multiset
{
  using key_type = Key;
  using value_type = Key;
  using key_compare = Compare;
  using value_compare = Compare;

  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = std::allocator_traits<Alloc>::pointer;
  using const_pointer = std::allocator_traits<Alloc>::const_pointer;

  ....
}
```

	Member function	Complexity
Element access	-	-
Capacity	empty	O(1)
	size, max_size, bucket_count	O(1)
	reserve	O(n) – $O(n^2)$
Modifiers	clear	O(n)
	erase	O(1) – O(n)
	insert, emplace	O(1) – O(n)
	emplace_hint	O(1) – O(n)
	swap	O(1)
	merge	O(n)
	extract	O(1) – O(n)
Lookup	count, find, contains, equal_range	O(1) – O(n)
Observers	key_eq, hash_function	O(1)
Allocator	get_allocator	O(1)

```cpp
#include <unordered_set>

void foo()
{
  std::unordered_set<int> s1 { 0, 1, 1, 2, 6, 3, 3, 4, 9, 5, 6, 7, 8, 9 };

  // s1 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

  std::unordered_set<int> s2;
  if (auto node = s1.extract(0))
  {
    s2.insert(node);

    // s1 { 1, 2, 3, 4, 5, 6, 7, 8, 9  }
    // s2 { 0 }
  }
}
```

std::unordered_map / std::unoredered_multimap

```
#include <unordered_map> // or <unordered_multimap>

template <class Key, class T,
          class Hash = std::hash<Key>,
          class KeyEqual = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T>>>
class unordered_map // or unordered_multimap
{
  using key_type = Key;
  using mapped_type = T;
  using value_type = std::pair<const Key, T>;
  using key_compare = Compare;
  using value_compare = Compare;
  using hasher = Hash;
  using key_equal = KeyEqual;

  using allocator_type = Alloc;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  ....
}
```

	Member function	Complexity
Element access	at, operator[]	O(1) – O(n)
Capacity	empty	O(1)
	size, max_size, bucket_count	O(1)
	reserve	O(n) – $O(n^2)$
Modifiers	clear	O(n)
	erase	O(1) – O(n)
	insert, insert_or_assign, emplace, try_emplace	O(1) – O(n)
	emplace_hint	O(1) – O(n)
	swap	O(1)
	merge	O(n)
	extract	O(1) – O(n)
Lookup	count, find, contains, equal_range	O(1) – O(n)
Observers	key_eq, hash_function	O(1)
Allocator	get_allocator	O(1)

```cpp
#include <string>
#include <unordered_map>

void foo()
{
  std::unordered_map<std::string, size_t> dict { { "foo", 0 },
                                                 { "bar", 1 },
                                                 { "foobar", 2 } };

  auto val = dict["bar"]; // find key "bar" and return mapped value 1
  dict["barfoo"] = 3;     // insert new element std::pair { "barfoo", 3 }
}
```

Adapters

std::stack

```
#include <stack>

template <class T,
          class Container = std::deque<T>>
class stack
{
  using container_type = Container;
  using value_type = typename Container::value_type;
  using size_type = typename Container::size_type;
  using reference = typename Container::reference;
  using const_reference = Container::const_reference;

  Container c;
  ....
}
```

	Member function	Complexity
Element access	top	O(1)
Capacity	empty	O(1)
	size	O(1)
Modifiers	swap	Same as underlying container
	push, emplace	Same as Container::push_back, Container::emplace_back
	pop	Same as Container::pop_back

```cpp
#include <stack>

void foo()
{
  std::stack<int> c1;
  c1.push(0);   // c1 { 0 }
  c1.push(10);  // c1 { 0, 10 }
  c1.push(100); // c1 { 0, 10, 100 }

  c1.pop(); // c1 { 0, 10 }
  std::cout << c1.top() << '\n'; // prints '10'
}
```

std::queue

```
#include <queue>

template <class T,
          class Container = std::deque<T>>
class queue
{
  using container_type = Container;
  using value_type = typename Container::value_type;
  using size_type = typename Container::size_type;
  using reference = typename Container::reference;
  using const_reference = Container::const_reference;

  Container c;
  ....
}
```

	Member function	Complexity
Element access	front, back	O(1)
Capacity	empty	O(1)
	size	O(1)
Modifiers	swap	Same as underlying container
	push, emplace	Same as Container::push_back, Container::emplace_back
	pop	Same as Container::pop_front

```
#include <queue>

void foo()
{
  std::queue<int> c1;
  c1.push(0);   // c1 { 0 }
  c1.push(10);  // c1 { 0, 10 }
  c1.push(100); // c1 { 0, 10, 100 }

  c1.pop(); // c1 { 10, 100 }
  std::cout << c1.front() << '\n'; // prints '10'
}
```

std::priority_queue

```
#include <queue>

template <class T,
          class Container = std::vector<T>,
          class Compare = std::less<typename Container::value_type>
class priority_queue
{
  using container_type = Container;
  using value_type = typename Container::value_type;
  using size_type = typename Container::size_type;
  using reference = typename Container::reference;
  using const_reference = Container::const_reference;

  Container c;
  ....
}
```

	Member function	Complexity
Element access	top	O(1)
Capacity	empty	O(1)
	size	O(1)
Modifiers	swap	Same as underlying container
	push, emplace	Same as Container::push_back, Container::emplace_back
	pop	Same as Container::pop_back

```
#include <queue>

void foo()
{
  std::priority_queue<int> pq;
  pq.push(3); // pq { 3 }
  pq.push(1); // pq { 3 1 }
  pq.push(5); // pq { 5 3 1 }
  pq.push(2); // pq { 5 3 2 1 }

  while (!c3.empty())
  {
    std::cout << c3.top() << '\n';
    c3.pop();
  }

  // prints '5' '3' '2' '1'
}
```

Iterators library

```cpp
#include <vector>

void foo()
{
  std::vector<int> v { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  for (int i = 0; i < v.size(); ++i) // bad practice, may not work on x64
    // some manipulation with v[i]
}
```

```cpp
#include <vector>

void foo()
{
  std::vector<int> v { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  for (int i = 0; i < v.size(); ++i) // bad practice, may not work on x64
    // some manipulation with v[i]

  for (size_t i = 0; i < v.size(); ++i) // better
    // some manipulation with v[i]
}
```

```cpp
#include <vector>

void foo()
{
  std::vector<int> v { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  for (int i = 0; i < v.size(); ++i) // bad practice, may not work on x64
    // some manipulation with v[i]

  for (size_t i = 0; i < v.size(); ++i) // better
    // some manipulation with v[i]

  for (auto curr = v.data(), end = start + v.size(); start != end; ++curr)
    // some manipulation with *curr
}
```

```cpp
#include <vector>
#include <list>

void foo()
{
  std::vector<int> v { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  std::list<int> l { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  for (auto curr = v.data(), end = start + v.size(); start != end; ++curr)
    // some manipulation with *curr

  // DOESN'T WORK!!!
  for (auto curr = &l.front(), end = curr + l.size(); curr != end; ++curr)
    // some manipulation with *curr
}
```

Iterator types

InputIterator	std::istream
ForwardIterator	std::forward_list, std::unordered_set, std::unordered_multiset, std::unordered_map std::unordered_multimap
BidirectionalIterator	std::list, std::set, std::multiset std::map, std::multimap
RandomAccessIterator ContiguousIterator	std::array, std::vector, std::string, std::string_view, std::valarray
OutputIterator	std::ostream

InputIterator

```
#include <sstream>
#include <iterator>
#include <string>

void foo(std::istream &is)
{
  std::istreambuf_iterator it { is };           // since C++17
  const std::istreambuf_iterator<char> end;

  const size_t count = std::distance(it, end); // O(n)
  std::string buf;
  buf.reserve(count);
  buf.assign(it, end); // buf == ""???
  ....
}

void bar()
{
  foo(std::istringstream { "Hello, world" });
}
```

InputIterator

```
#include <sstream>
#include <iterator>
#include <string>

void foo(std::istream &is)
{
  std::istreambuf_iterator it { is };        // since C++17
  const std::istreambuf_iterator<char> end;

  std::string buf { it, end }; // ok, buf == "Hello, World"
  ....
}

void bar()
{
  foo(std::istringstream { "Hello, world" });
}
```

InputIterator

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

void foo(std::istream &is)
{
  std::vector<int> v { std::istream_iterator<int> { is },
                       std::istream_iterator<int> {} };
}

void bar()
{
  foo(std::cin);
}
```

ForwardIterator

```
#include <forward_list>
#include <vector>

void foo(const std::forward_list<int> &l)
{
  const size_t size = std::distance(l.begin(), l.end()); // O(n)

  std::vector<int> v1 { l.begin(), l.end() };
  std::vector<int> v2 { l.begin(), l.end() };


  auto it = l.begin();
  ++it;
  it = std::next(it, 3); // or std::advance(it, 3)
  --it; // compile-time error
}
```

BidirectionalIterator

```
#include <list>
#include <vector>

void foo(const std::list<int> &l)
{
  const size_t size = std::distance(l.begin(), l.end()); // O(n)

  std::vector<int> v1 { l.begin(), l.end() };   // original order
  std::vector<int> v2 { l.rbegin(), l.rend() }; // reverse order

  auto it = l.begin();
  ++it; // ok
  --it; // ok

  it = it + 5; // compile-time error
  it = std::next(5); // or std::advance(it, 5);
  it = std::prev(5); // or std::advance(it, -5);
}
```

BidirectionalIterator

```
#include <string>

std::string reverse_string(const std::string &str)
{
  return { str.rbegin(), str.rend() };
}

std::string reverse_string(std::string str)
{
  std::reverse(str.begin(), str.end());
  return str;
}
```

RandomAccessIterator / ContiguousIterator

```
#include <list>
#include <vector>

void foo(const std::vector<int> &v)
{
  const size_t size = std::distance(l.begin(), l.end()); // O(1)

  std::vector<int> v1 { l.begin(), l.end() };   // original order
  std::vector<int> v2 { l.rbegin(), l.rend() }; // reverse order

  auto it = l.begin();
  ++it; // ok
  --it; // ok

  it = it + 5; // ok
  it = std::next(5); // or std::advance(it, 5);
  it = std::prev(5); // or std::advance(it, -5);
}
```

OutputIterator

```
#include <sstream>
#include <iterator>
#include <string>

void foo(std::ostream &is)
{
  std::string buf { "Hello, World" };

  std::copy(buf.begin(), buf.end(), std::ostreambuf_iterator { is });

  // Prints "Hello, World"
}

void bar()
{
  foo(std::cout);
}
```

OutputIterator

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

void foo(std::ostream &is)
{
  std::vector<int> v { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  std::copy(v.begin(), v.end(), std::ostream_iterator<int> { is });

  // Prints "0 1 2 3 4 5 6 7 8 9"
}

void bar()
{
  foo(std::cout);
}
```

std::iterator_traits

```
template <class InputIt>
ptrdiff_t distance(InputIt first, InputIt last)
{
  ptrdiff_t dist = 0;
  for (; first != last; ++first, ++dist);

  return dist;
}
```

```
// <iterator>

template <class Iter>
struct iterator_traits
{
  using difference_type   = typename Iter::difference_type;
  using value_type        = typename Iter::value_type;
  using pointer           = typename Iter::pointer;
  using reference         = typename Iter::reference;
  using iterator_category = typename Iter::iterator_category;
};

template <class T>
struct iterator_traits<T*>
{
  using difference_type   = ptrdiff_t
  using value_type        = T;
  using pointer           = T*;
  using reference         = T&;
  using iterator_category = random_access_iterator_tag;
};
```

```
// <type_traits>

struct input_iterator_tag {};

struct forward_iterator_tag : input_iterator_tag {};

struct bidirectional_iterator_tag : forward_iterator_tag {};

struct random_access_iterator_tag : bidirectional_iterator_tag {};

struct contiguous_iterator_tag : random_access_iterator_tag {};

struct output_iterator_tag {};
```

```
template <typename It>
using it_cat = typename std::iterator_traits<It>::iterator_category;

template <typename It>
using it_diff = typename std::iterator_traits<It>::difference_type;

template <class It>
auto distance(It first, It last, std::input_iterator_tag)
{
  it_diff<It> dist = 0;
  for (; first != last; ++first, ++dist);

  return dist;
}

template <class It>
auto distance(It first, It last, std::random_access_iterator_tag)
{
  return first < last ? (last - first) : -(first - last);
}

template <class It>
auto distance(It first, It last)
{
  return distance(first, last, it_cat<It> {});
}
```

END

Q&A