std::allocator<T>::rebind<U>

- В std::allocator – это member type, являющийся struct
- Нужен для получения аллокатора других объектов U из аллокатора объектов типа T.
- Он нам понадобится позже, для имплементации std::list.

```
int main () {
    std::allocator<int> initial;
    int* first = initial.allocate( n: 10);
    initial.deallocate(first,  n: 10);

    decltype(initial)::rebind<double>::other other;
    double* second = other.allocate( n: 10);
    other.deallocate(second,  n: 10);
}
```

В примере изначальный std::allocator, предназначенный для выделения int-ов был использован для получения другого инстанса std::allocator’a, который позволяет выделять double.

std::allocator<T>::rebind<U>

- В std::allocator rebind выглядит как член-тип являющийся структурой с единственным шаблонным полем other типа std::allocator<U>.

```
template <typename T>
class allocator {
public:
    /// ...

    template<typename U>
    struct rebind {
        typedef allocator<U> other;
    };

    ///...
};
```

Определить rebind для своего аллокатора по аналогии с std::allocator не трудно: общий вид определения приведен на картинке.

Такая реализация вызывает конструктор по умолчанию другого инстанса аллокатора.

Но что если мы не определим своего rebind и передадим его в какой-нибудь STL класс, который его вызывает? Ошибка компиляции?

std::allocator_traits<Allocator>

- В C++ 11 есть такая умная структура как std::allocator_traits<Allocator>. Она позволяет доопределить стандартным поведением методы вашего аллокатора, если каких-то методов не хватает.
- Все STL-контейнеры, требующие Allocator будут вызывать его методы не напрямую, а строго через allocator_traits<Allocator>. Это позволит быть уверенным, что любой аллокатор с методами allocate, deallocate и членом-типом value_type будет доопределен стандартным поведением.

std::allocator_traits<Allocator>

```cpp
template <typename T>
struct Alloc {
public:
    using value_type = T;

    T* allocate (const size_t n) {
        cout << "alloc " << n << endl;
        return reinterpret_cast<T*>(new char[n * sizeof(T)]);
    };

    void deallocate (T* p, const size_t n) {
        cout << "dealloc " << n << endl;
        delete[] reinterpret_cast<char*>(p);
    };
};

int main () {
    Alloc<int> alloc;
    /// Not working:
    decltype(alloc)::rebind<double>::other other_alloc;

    /// But...
    allocator_traits<decltype(alloc)>::rebind_alloc<double> other_alloc;
    other_alloc.deallocate(other_alloc.allocate( n: 10),  n: 10);
}
```

```
pres
alloc 10
dealloc 10
```

STL Containers: обзор

Определение по C++ reference: A ***Container*** is an object used to store other objects and taking care of the management of the memory used by the objects it contains.

Ввиду определения, очевидно, что контейнеры работают через аллокаторы.
Есть 4 большие группы контейнеров, о каждой из них поговорим по-отдельности.

Последовательные	Ассоциативные (упорядоченные)	Ассоциативные (неупорядоченные)	Адаптеры
array	set	unordered_set	stack
vector	multiset	unordered_multiset	queue
deque	map	unordered_map	priority_queue
list	multimap	unordered_multimap	
forward_list			

STL Containers: array

Фиксированный нерасширяемый массив. Грубо говоря, аналог C-style-array.
Не использует аллокатор, array владеет элементами, память выделяется внутри
без аллокатора!

- Шаблонные параметры: <typename T, size_t N>.

Доступ к элементам:

- operator[] **(const и non-const)**
- front() **(const и non-const)**
- back() **(const и non-const)**
- at() **(const и non-const)**
- data() **(const и non-const)**

Размер:

- empty()
- size()
- max_size()

Операции:

- fill()
- swap()

Сравнение содержимого:

- Operator <
- Operator >
- Operator <=
- Operator >=
- Operator ==
- Operator !=

(Все кроме operator== убраны с C++20)

STL Containers: array

operator[] VS .at()

- И первый и второй имеют константные и неконстантные спецификации. В первом случае оба метода – это getter-ы элементов, во втором – и setter-ы и getter-ы (то есть accessor-ы). Если бы не было константной специализации – на константном массиве нельзя было бы вызывать operator[] даже как getter.
- Поведение в случае корректного входного аргумента (индекса) совпадает.
- Оба метода возвращают ссылки на элементы.
- Однако, если методу .at() передать индексу-аргумент, превышающий .size() – 1, то вылетит ошибка std::out_of_range. operator[] в этом случае ошибки не выдает!
- Но почему [] тогда не помечен noexcept? Потому что может произойти UB.

STL Containers: array

size() VS max_size()

- Первый метод возвращает текущее кол-во элементов в контейнере.
- Второй – максимально возможное количество элементов в контейнере **данного типа.** Это число обычно связано с различными ограничениями конкретных имплементаций библиотек (разные имплементации могут позволять хранить разное максимальное кол-во элементов).
- В случае std::array оба метода возвращают compile-time константу равную шаблонному аргументу N.

STL Containers: array

Сравнение содержимого

- operator== и operator!= проверяют точное соответствие элементов поочередно применяя соответствующий оператор к элементам внутри массива.
- Остальные операторы сравнивают элементы в лексикографическом порядке. Сравнение производит функция эквивалентная std::lexicographical_compare
- Начиная с C++20 все из шести вышеперечисленных операторов кроме == были заменены на оператор <=>, который, впрочем, обсуждать сейчас не будем.

STL Containers: array

Еще кое-что внутри std::array…

Итераторы

Member types

Member type	Definition
value_type	T
size_type	std::size_t
difference_type	std::ptrdiff_t
reference	value_type&
const_reference	const value_type&
pointer	value_type*
const_pointer	const value_type*
iterator	*LegacyRandomAccessIterator* and *ConstexprIterator* (since C++20) that is a *LiteralType* (since C++17)
const_iterator	Constant *LegacyRandomAccessIterator* and *ConstexprIterator* (since C++20) that is a *LiteralType* (since C++17)
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

Iterators

begin **cbegin**	returns an iterator to the beginning (public member function)
end **cend**	returns an iterator to the end (public member function)
rbegin **crbegin**	returns a reverse iterator to the beginning (public member function)
rend **crend**	returns a reverse iterator to the end (public member function)

STL Containers: vector

Динамический расширяемый массив. Амортизированное время работы основных операций – O(1). На курсе алгоритмов должны были рассказывать как он работает с точки зрения алгоритмов.

- Шаблонные параметры: <typename T, typename Allocator = std::allocator<T>>

Доступ к элементам:

- operator[] **(const и non-const)**
- front() **(const и non-const)**
- back() **(const и non-const)**
- at() **(const и non-const)**
- data() **(const и non-const)**

Размер:

- empty()
- size()
- max_size()
- reserve()
- capacity()
- shrink_to_fit()
- resize()

Операции:

- assign()
- swap()
- get_allocator()

Сравнение содержимого:

- Operator <
- Operator >
- Operator <=
- Operator >=
- Operator ==
- Operator !=

(Все кроме operator== убраны с C++20)

STL Containers: vector

Методы управления размером

- resize(N, el?) – производит расширение вектора до размера N, причем если необходимо добавить новые элементы и опциональный аргумент el указан – будут созданы копии элемента el. Если же el не указан, то в этой ситуации будет вызван конструктор T по умолчанию.
- size() – возвращает текущее количество элементов
- capacity() – возвращает текущий размер буффера

STL Containers: vector

Методы управления размером, продолжение

- max_size() – в отличии от std::array – возвращает число отличное от size(), которое действительно обозначает максимальное кол-во элементов которое можно записать в вектор.
- reserve(N) – в случае если N > capacity(), – этот метод резервирует для вектора ровно N ячеек, таким образом новый capacity() становится ровно N. Если N > max_size(), то std::length_error. Сложность – O(size()).
- shrink_to_fit() – урезает буффер и делает capacity() равным size().

STL Containers: vector

Модификаторы класса std::vector

- push_back(), emplace_back()
- pop_back()
- insert(), emplace()
- erase()

Emplace – это своего рода construct у аллоктора. То есть emplace позволяет передать необходимые параметры через аргументы непосредственно для вызова конструктора. Таким образом не придется создавать копии объекта.

STL Containers: vector

О методах push_back и pop_back

- Добавляют и соответственно удаляют элементы в массив.
- При необходимости – push_back производит расширение буффера в два раза, переаллоцируя (с помощью allocator_traits<Allocator>) весь выделенный кусок памяти.
- Метод pop_back же производит удаление (деструктурирование) элемента из массива. При возможности – происходит сужение массива в два раза. Лишняя память деаллоцируется (с помощью allocator_traits<Allocator>)

STL Containers: vector

О методах insert и erase

- Работают с так называемыми итераторами, принимая на вход итератор на элемент и производя вставку со сдвигом хвоста до места на которое этот итератор указывает, либо его удаление со сдвигом хвоста соответственно.
- Асимптотика: O(n - i), так как надо сдвинуть “хвост”.

STL Containers: vector

О методе assign

- Имеет три перегрузки, но мы разберем две:
- 1) void assign(size_type count, const T& value) – заменяет контент внутри контейнера на ровно count копий элемента value.
- 2) void assign(Iter first, Iter last) – заменяет контент внутри контейнера на диапазон [first, last). Однако, если итераторы указывали на элементы данного вектора – получим UB.

STL Containers: vector

О копировании аллокатора.

- Что делать при конструкторе копирования с аллокатором!?
- Есть метод select_on_container_copy_construction()
- И не только он =)

Influence on container operations

Expression	Return type	Description
`a.select_on_container_copy_construction()` (optional)	A	• Provides an instance of A to be used by the container that is copy-constructed from the one that uses a currently. • (Usually returns either a copy of a or a default-constructed A.)
Type-id	**Aliased type**	**Description**
`A::propagate_on_container_copy_assignment` (optional)	`std::true_type` or `std::false_type` or derived from such.	• `std::true_type` or derived from it if the allocator of type A needs to be copied when the container that uses it is copy-assigned. • If this member is `std::true_type` or derived from it, then A must satisfy *CopyAssignable* and the copy operation must not throw exceptions. • Note that if the allocators of the source and the target containers do not compare equal, copy assignment has to deallocate the target's memory using the old allocator and then allocate it using the new allocator before copying the elements (and the allocator).
`A::propagate_on_container_move_assignment` (optional)		• `std::true_type` or derived from it if the allocator of type A needs to be moved when the container that uses it is move-assigned. • If this member is `std::true_type` or derived from it, then A must satisfy *MoveAssignable* and the move operation must not throw exceptions. • If this member is not provided or derived from `std::false_type` and the allocators of the source and the target containers do not compare equal, move assignment cannot take ownership of the source memory and must move-assign or move-construct the elements individually, resizing its own memory as needed.
`A::propagate_on_container_swap` (optional)		• `std::true_type` or derived from it if the allocators of type A need to be swapped when two containers that use them are swapped. • If this member is `std::true_type` or derived from it, lvalues of A must be *Swappable* and the swap operation must not throw exceptions. • If this member is not provided or derived from `std::false_type` and the allocators of the two containers do not compare equal, the behavior of container swap is undefined.

STL Containers: vector

.swap() и зачем он нужен

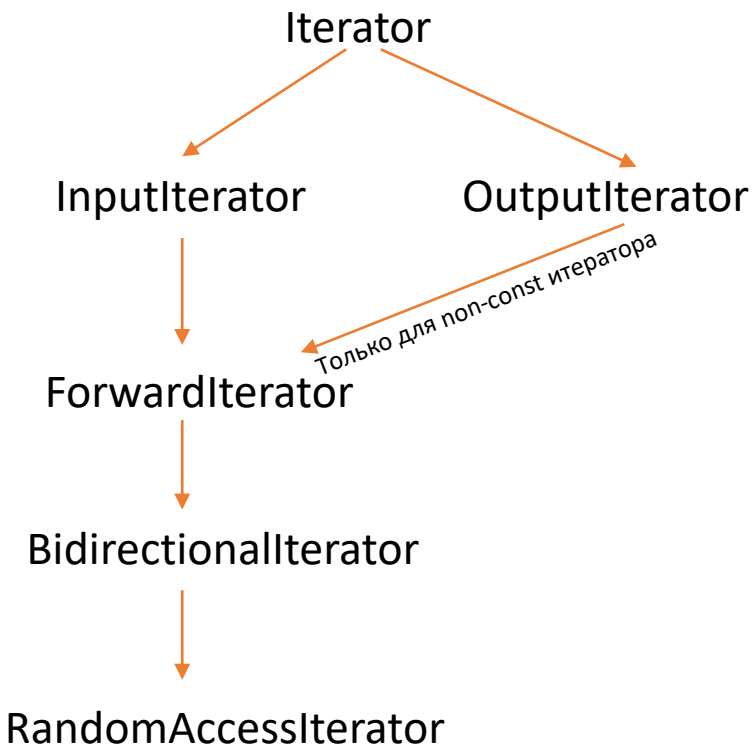
- Обычный swap работал бы за O(N), так как вызывал бы конструктор копирования массива, создавая лишнюю копию.
- Специализированный swap позволяет просто перекинуть указатели за O(1). Он не только присутствует внутри класса, но и вынесен в виде специализации std::swap для std::vector<T, N>.
- По аналогии с конструктором копирования, для swap-а аллокторов можно заимплементить propagate_on_container_swap.

```
template <typename T, typename Allocator = std::allocator<T>>
class vector {
private:
    T* _arr;
    Allocator _alloc;
    size_t _size;
    size_t _capacity;

public:
    /// ...
    void swap (vector &other) noexcept(noexcept(
        std::allocator_traits<Allocator>::propagate_on_container_swap::value ||
        std::allocator_traits<Allocator>::is_always_equal::value))
    {
        std::swap(_arr, other._arr);
        std::swap(_size, other._size);
        std::swap(_capacity, other._capacity);

        if (std::allocator_traits<Allocator>::propagate_on_container_swap::value) {
            std::cout << "swapped" << std::endl;
            std::swap(_alloc, other._alloc);
        }
    }
    /// ...
};
```

Iterators: общие слова

- Итератор – некий аналог pointer-а, однако это не совсем верное сравнение, потому что большинство итераторов запрещают некоторые операции, которые, однако, были бы доступны для pointer-а.
- Итератор – внутренняя публичная структура во всех контейнерах. К ней можно обратиться с оператором ::. Например: vector<int>::iterator iter = …
- Во всех контейнерах есть специальные методы, которые позволяют получить итератор на начало: .begin().

Iterators: виды



Requirements

The type `It` satisfies *LegacyIterator* if

- The type `It` satisfies *CopyConstructible*, and
- The type `It` satisfies *CopyAssignable*, and
- The type `It` satisfies *Destructible*, and
- lvalues of type `It` satisfy *Swappable*, and
- `std::iterator_traits<It>` has member typedefs `value_type`, `difference_type`, `reference`, `pointer`, and `iterator_category` , and
- Given `r`, an lvalue of type `It`, the following expressions must be valid and have their specified effects:

Expression	Return Type	Precondition
`*r`	unspecified	**r** is *dereferenceable* (see below)
`++r`	`It&`	**r** is *incrementable* (the behavior of the expression `++r` is defined)

Важно понимать что все перечисленные виды итераторов – НЕ являются классами, стрелки НЕ показывают наследования.

На самом деле, эта классификация сугубо концептуальна и нужна лишь для понимания накладываемых ограничений на соответствующий итератор.

Стрелка от a к b обозначает, что любой итератор типа b является итератором типа a.

Iterators: Input Iterator

- InputIterator - это Iterator, который может читать из элемента на который указывает. InputIterator гарантирует валидность только для однопроходных алгоритмов: после инкрементирования InputIterator – все копии его предыдущего значения могут стать недействительными.

Requirements

The type `It` satisfies *LegacyInputIterator* if

- The type `It` satisfies *LegacyIterator*
- The type `It` satisfies *EqualityComparable*

And, given

- `i` and `j`, values of type `It` or `const It`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`
- `value_type`, the type denoted by `std::iterator_traits<It>::value_type`

The following expressions must be valid and have their specified effects

Expression	Return	Equivalent expression	Notes
`i != j`	contextually convertible to `bool`	`!(i == j)`	**Precondition**: `(i, j)` is in the domain of `==`.
`*i`	`reference`, convertible to `value_type`	If `i == j` and `(i, j)` is in the domain of `==` then this is equivalent to `*j`.	**Precondition**: `i` is dereferenceable. The expression `(void)*i, *i` is equivalent to `*i`.
`i->m`		`(*i).m`	**Precondition**: `i` is dereferenceable.
`++i`	`It&`		**Precondition**: `i` is dereferenceable. **Postcondition**: `i` is dereferenceable or `i` is past-the-end. **Postcondition**: Any copies of the previous value of `i` are no longer required to be either dereferenceable or to be in the domain of `==`.
`(void)i++`		`(void)++i`	

`*i++`	convertible to `value_type`	`value_type x = *i;`
		`++i;`
		`return x;`	

Iterators: Output Iterator

- OutputIterator - это Iterator, который может писать в элемент на который указывает. OutputIterator гарантирует валидность только для однопроходных алгоритмов: после инкрементирования OutputIterator – все копии его предыдущего значения могут стать недействительными.

Requirements

The type X satisfies *LegacyOutputIterator* if

- The type X satisfies *LegacyIterator*
- X is a class type or a pointer type

And, given

- o, a value of some type that is writable to the output iterator (there may be multiple types that are writable, e.g. if `operator=` may be a template. There is no notion of `value_type` as for the input iterators)
- r, an lvalue of type X,

The following expressions must be valid and have their specified effects

Expression	Return	Equivalent expression	Pre-condition	Post-conditions	Notes
`*r = o`	(not used)		r is dereferenceable	r is incrementable	After this operation r is not required to be dereferenceable and any copies of the previous value of r are no longer required to be dereferenceable or incrementable.
`++r`	`X&`		r is incrementable	`r` and `++r` designate the same iterator object, r is dereferenceable or past-the-end	After this operation r is not required to be incrementable and any copies of the previous value of r are no longer required to be dereferenceable or incrementable.

`r++`	convertible to `const X&`	`X temp = r;`
		`++r;`
		`return temp;`			

`*r++ = o`	(not used)	`*r = o;` `++r;`			

Iterators: Forward Iterator

- ForwardIterator - это InputIterator который позволяет итерироваться только в одном направлении. ForwardIterator гарантирует валидность даже для многопроходных алгоритмов. Если инстанс ForwardIterator – не константный, то он является OutputIterator'ом.

Requirements

The type `It` satisfies *LegacyForwardIterator* if

- The type `It` satisfies *LegacyInputIterator*
- The type `It` satisfies *DefaultConstructible*
- Objects of the type `It` provide *multipass guarantee* described below
- The type `std::iterator_traits<It>::reference` must be exactly
  - T& if `It` satisfies *LegacyOutputIterator* (`It` is mutable)
  - `const` T& otherwise (`It` is constant),

  (where T is the type denoted by `std::iterator_traits<It>::value_type`)
- Equality and inequality comparison is defined over all iterators for the same underlying sequence and the value initialized-iterators (since C++14).

And, given

- `i`, dereferenceable iterator of type `It`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`

The following expressions must be valid and have their specified effects

Expression	Return type	Equivalent expression	Notes
`i++`	`It`	`It ip=i; ++i; return ip;`	
`*i++`	`reference`		

A *mutable* *LegacyForwardIterator* is a *LegacyForwardIterator* that additionally satisfies the *LegacyOutputIterator* requirements.

Iterators: Bidirectional Iterator

- BidirectionalIterator - это ForwardIterator который позволяет итерироваться по контейнеру в двух направлениях.

Requirements

The type `It` satisfies *LegacyBidirectionalIterator* if

- The type `It` satisfies *LegacyForwardIterator*

And, given

- `a` and `b`, iterators of type `It`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`

The following expressions must be valid and have their specified effects

Expression	Return	Equivalent expression	Notes
`--a`	`It&`		Preconditions: • a is decrementable (there exists such `b` that `a == ++b`) Postconditions: • a is dereferenceable • `--(++a) == a` • If `--a == --b` then `a == b` • `a` and `--a` designate the same iterator object

`a--`	convertible to `const It&`	`It temp = a;`
		`--a;`
		`return temp;`	

`*a--`	`reference`		

A *mutable* *LegacyBidirectionalIterator* is a *LegacyBidirectionalIterator* that additionally satisfies the *LegacyOutputIterator* requirements.

Iterators: Random Access Iterator

- RandomAccessIterator - это BidirectionalIterator который может быть перемещен в любую позицию за константное время.

Requirements

The type `It` satisfies *LegacyRandomAccessIterator* if

- The type `It` satisfies *LegacyBidirectionalIterator*

And, given

- `value_type`, the type denoted by `std::iterator_traits<It>::value_type`
- `difference_type`, the type denoted by `std::iterator_traits<It>::difference_type`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`
- `i`, `a`, `b`, objects of type `It` or `const It`
- `r`, a value of type `It&`
- `n`, an integer of type `difference_type`

The following expressions must be valid and have their specified effects

Expression	Return type	Operational semantics	Notes

`r += n`	`It&`	`difference_type m = n;`
		`if (m >= 0) while (m--) ++r;`
		`else while (m++) --r;`
		`return r;`
			• **n** can be both positive or negative
			• The complexity is constant (that is, the implementation cannot actually execute the while loop shown in operational semantics)

`a + n` `n + a`	`It`	`It temp = a;` `return temp += n;`	• **n** can be both positive or negative • `a + n == n + a`
`r -= n`	`It&`	`return r += -n;`	The absolute value of `n` must be within the range of representable values of `difference_type`.
`i - n`	`It`	`It temp = i;` `return temp -= n;`	
`b - a`	`difference_type`	`return n;`	Precondition: • there exists a value `n` of type `difference_type` such that `a+n==b` Postcondition: • `b == a + (b - a)`.
`i[n]`	convertible to `reference`	`*(i + n)`	
`a < b`	contextually convertible to `bool`	`b - a > 0`	Strict total ordering relation: • **`!(a < a)`** • if **`a < b`** then **`!(b < a)`** • if **`a < b`** and **`b < c`** then **`a < c`** • **`a < b`** or **`b < a`** or **`a == b`** (exactly one of the expressions is true)
`a > b`	contextually convertible to `bool`	`b < a`	Total ordering relation opposite to `a < b`
`a >= b`	contextually convertible to `bool`	`!(a < b)`	
`a <= b`	contextually convertible to `bool`	`!(a > b)`	

Iterators: const и reverse

Итераторы могут быть константными и обратными. Таким образом получаем 4 вида итераторов:

- Iterator
- const_iterator
- reverse_iterator
- const_reverse_iterator

- Const итератор может быть модифицирован сам, однако с помощью него нельзя модифицировать элементы внутри контейнера. Аналог – указатель на константу.
- Reverse итератор – просто итератор в обратном направлении.

У всех поледовательных контейнеров есть методы:

- begin(), end()
- cbegin(), cend()
- rbegin(), rend(),
- crbegin(), crend()

Эти методы нужны для получения итератора на первый элемент и на мнимый элемент, следующий за последним.

```
int main () {
    vector<int> vec( il: {1, 2, 3, 4, 5});

    for (vector<int>::const_iterator i = vec.cbegin(); i != vec.cend(); ++ i)
        cout << *i << " ";
    cout << endl;

    for (vector<int>::const_reverse_iterator i = vec.crbegin(); i != vec.crend(); ++ i)
        cout << *i << " ";
    cout << endl;
}
```

```
/Users/Costello/Desktop/pres/cmake-build-debug/pres
1 2 3 4 5
5 4 3 2 1

Process finished with exit code 0
```

Iterators: std::reverse_iterator<…>

- Неужели если мы захотим написать свой собственный vector, нам придется писать 4 внутренних класса для каждого итератора… Нет! Ведь есть std::reverse_iterator<Iterator>. Этот класс принимает на вход темплейтным аргументом обычный итератор, и изменяет его направление.
- Чтобы еще раз не делать reverse на копии обратного итератора – можно воспользоваться методом base, который вернет базовый итератор.
- Очевидно, сам std::reverse_iterator является обычным адаптером.

Iterators: std::iterator<...>

- До C++17 был актуален темплейтный класс std::iterator, от которого публично наследовались итераторы всех контейнеров и в том числе std::reverse_iterator.

std::iterator

Defined in header <iterator>

```
template<
    class Category,
    class T,
    class Distance = std::ptrdiff_t,     (deprecated in C++17)
    class Pointer = T*,
    class Reference = T&
> struct iterator;
```

В темплейтных аргументах для такого итератора нужно в том числе передать так называемый tag. Этот тег нужнен для того, чтобы уточнить какой это именно итератор.

Defined in header <iterator>

```
struct input_iterator_tag { };                                          (1)
struct output_iterator_tag { };                                         (2)
struct forward_iterator_tag : public input_iterator_tag { };            (3)
struct bidirectional_iterator_tag : public forward_iterator_tag { };    (4)
struct random_access_iterator_tag : public bidirectional_iterator_tag { };  (5)
struct contiguous_iterator_tag: public random_access_iterator_tag { };  (6)  (since C++20)
```

Iterators: std::iterator_traits<…>

std::iterator_traits

Defined in header <iterator>

```
template< class Iter >
struct iterator_traits;
```

```
template< class T >
struct iterator_traits<T*>;
```

```
template< class T >
struct iterator_traits<const T*>;
```
(until C++20)

`std::iterator_traits` is the type trait class that provides uniform interface to the properties of *LegacyIterator* types. This makes it possible to implement algorithms only in terms of iterators.

The class defines the following types:

- `difference_type` - a signed integer 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.

The template can be specialized for user-defined iterators so that the information about the iterator can be retrieved even if the type does not provide the usual typedefs.

User specializations may define the member type `iterator_concept` to one of iterator category tags, to indicate conformance to the iterator concepts. (since C++20)

Template parameters

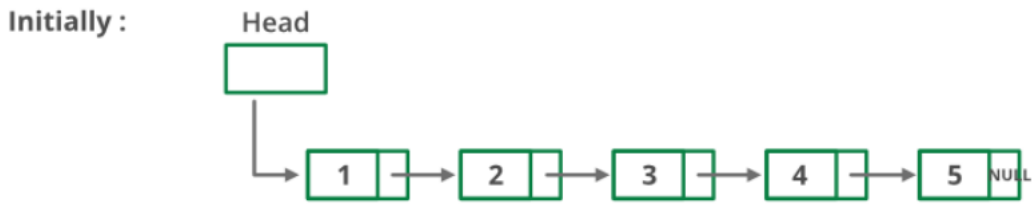
`Iter` - the iterator type to retrieve properties for

Member types

Member type	Definition
`difference_type`	`Iter::difference_type`
`value_type`	`Iter::value_type`
`pointer`	`Iter::pointer`
`reference`	`Iter::reference`
`iterator_category`	`Iter::iterator_category`

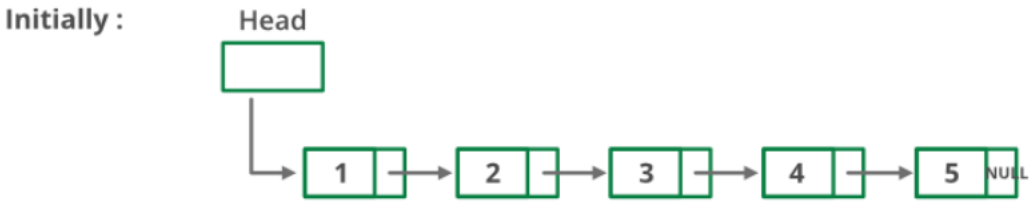
STL Containers: list

- Двунаправленный список. Имеет методы push_back, push_front, pop_front, pop_back. operator [] – отсутствует!
- List реализован с вспомогательным классом Node, который аллоцируется при помощи sarebindenного allocator’a.
- Класс итератора: Bidirectional



STL Containers: forward_list

- Однонаправленный список. Имеет методы push_front, pop_front. operator [] – отсутствует!
- List реализован с вспомогательным классом Node, который аллоцируется при помощи sarebindenного allocator’a.
- Класс итератора: Forward



PS: на картинке должны быть еще и обратные стрелочки.

STL Containers: deque

- Deque (Double Ended Queue) – класс, который умеет делать все операции, которые поддерживает list и vector.
- Учетное время работы основных операций push_back, push_front, pop_back, pop_front, operator [] – O(1).
- Тип итератора – RandomAccess
- Использует аллокатор: да.

STL Containers: stack

std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

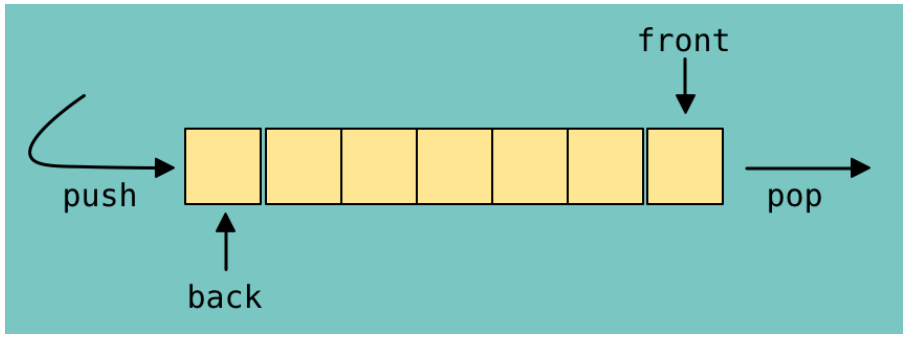
- Stack – обычный известный нам stack, доступны методы push и pop.
- Класс stack – это не отдельный класс, который реализует сам stack “подкапотно”, это – некий адаптер над классом Container (последний должен уметь делать все, что нужно stack-у).
- Container передается через template-ный аргумент.

STL Containers: queue

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```



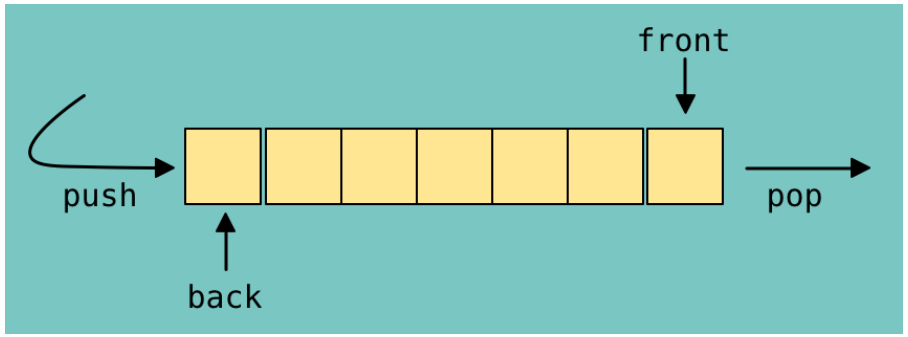
- queue – это очередь.
- Класс queue – это не отдельный класс, который реализует сам stack “подкапотно”, это – некий адаптер над классом Container (последний должен уметь делать все, что нужно queue).
- Container передается через template-ный аргумент.

STL Containers: queue

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```



- queue – это очередь.
- Класс queue – это не отдельный класс, который реализует сам stack “подкапотно”, это – некий адаптер над классом Container (последний должен уметь делать все, что нужно queue).
- Container передается через template-ный аргумент.

STL Containers: set

std::set

Defined in header <set>

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

- Set – ассоциативный упорядоченный контейнер, представляет идею множества элементов.
- Элементы упорядочены в set-е определенным отношением порядка, например less.
- Тип итератора: Bidirectional
- Подкапотно реализован на КЧ-деревьях
- Время работы – O(log(n)) на основные операции.

STL Containers: map

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

- Map – ассоциативный упорядоченный контейнер, представляет идею соответствия элементов множества Key и элементов множества T.
- Элементы упорядочены в map-е определенным отношением порядка, например less.
- Тип итератора: Bidirectional
- Подкапотно реализован на КЧ-деревьях
- operator[] – создает значение проинициализированное default-ным конструктором по ключу с которым вызван. Проверять наличие элемента в map-е лучше методом at.

STL Containers: unordered_... и multi_...

std::unordered_set

Defined in header <unordered_set>

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

std::unordered_map

Defined in header <unordered_map>

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

- Unordered_... – версии реализованные на hash—таблицах, учетное время O (1).
- Multi_... – версии которые предполагают возможность включения многих значений в контейнер.
- Стоит использовать именно unordered_... В продакшне, если предполагается много операций по добавлению элентов и хочется иметь возможность хранить много элементов.
- Если нужно использовать unordered_... Контейнеры для вашего типа – придется написать собственный hash.
- Потенциальная опасность – вызов rehash. Это происходит редко, но метко =) Операция требует много времени.
- Есть комбинированные версии: unordered_multi(map|set)