

# STL. Часть 2

Евгений Белоусов  
Ведущий программист в компании  
IQTech



# Проверка связи





## Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



## Поставьте в чат:

-  если меня видно и слышно
-  если нет

# Евгений Белоусов

О спикере:

- Ведущий программист в компании IQTech
- Работает в IT с 2011
- Опыт разработки на C++ более 12 лет



# Вспоминаем прошное занятие

**Вопрос:** как `std::vector` хранит свои элементы  
в памяти?



# Вспоминаем прошное занятие

**Вопрос:** как `std::vector` хранит свои элементы  
в памяти?

**Ответ:** непрерывно



# Вспоминаем прошное занятие

**Вопрос:** какой контейнер соответствует  
принципу LIFO (last in, first out)?



# Вспоминаем прошлые занятия

**Вопрос:** какой контейнер соответствует  
принципу LIFO (last in, first out)?

**Ответ:** `std::stack`



# Вспоминаем прошрое занятие

**Вопрос:** какой контейнер лучше подойдет, если необходимо хранить пары ключ-значение, порядок сортировки не важен?





# Вспоминаем прошрое занятие

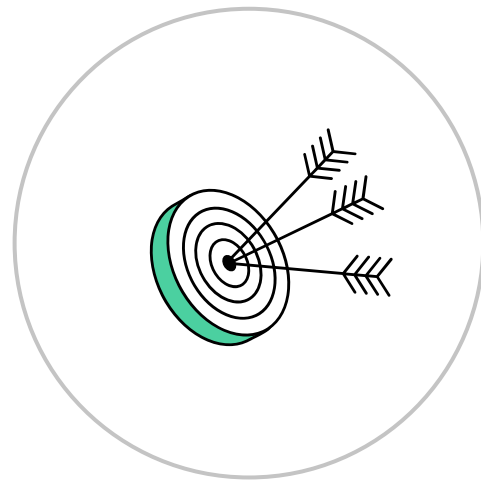
**Вопрос:** какой контейнер лучше подойдет, если необходимо хранить пары ключ-значение, порядок сортировки не важен?

**Ответ:** `std::unordered_map`



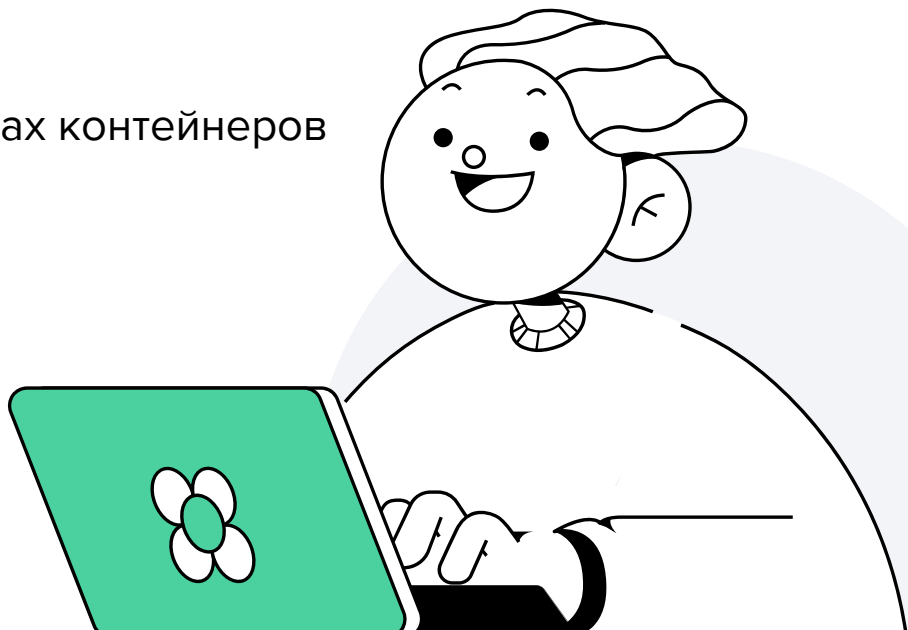
# Цели занятия

- Узнаем, что такое итератор
- Узнаем какие бывают итераторы и как с ними работать



# План занятия

- 1 Категории итераторов
- 2 Операции над итераторами
- 3 Примеры использования
- 4 Использование в алгоритмах и методах контейнеров
- 5 Домашнее задание



\*Нажми на нужный раздел для перехода

# А зачем это нужно?

Одной из задач при разработке библиотеки STL было разделение двух сущностей:

- контейнеров
- алгоритмов

Для взаимодействия алгоритма с данными контейнера пришлось ввести промежуточную сущность — **итератор**.

# А зачем это нужно?

Итераторы позволили алгоритмам получать доступ к данным, содержащимся в контейнере, независимо от типа контейнера:

- они знают о внутреннем устройстве контейнера
- предоставляют удобный интерфейс для работы с данными контейнера, будь то доступ к элементам или какой-то специальный обход всех элементов контейнера

Но для этого в каждом контейнере **потребовалось определить класс итератора**.

# А зачем это нужно?

Можно сказать, что итераторы — это “безопасные” указатели с дополнительным функционалом. Без них нам бы пришлось применять арифметику указателей.

```
int arr[] = { 1, 7, 14, -5 };  
auto elem = arr[2];  
auto elem1 = *(arr + 2); // сдвигаем указатель на 2 элемента
```

В данном случае сдвиг указателя срабатывает из-за того, что элементы массива располагаются непрерывно друг за другом в памяти.

У некоторых контейнеров `std`, такой сдвиг мог бы привести к неопределённому поведению потому, что элементы не располагаются друг за другом в памяти. Более того, у множества контейнеров оператор `[]` не перегружен.

# Категории итераторов



1



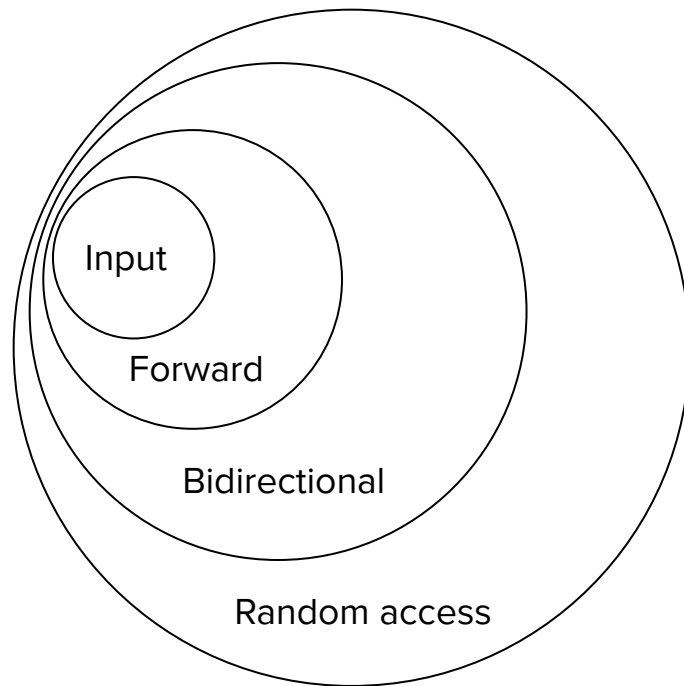
**Итератор — это объект, который способен перебирать элементы контейнера без необходимости пользователю знать его реализацию**



# Категории итераторов

- **Input** — итераторы, из которых можно читать. Поддерживают операцию чтения, инкремент без нескольких проходов
- **Forward** — добавляется поддержка инкремента с несколькими проходами
- **Bidirectional** — добавляется операция декремента
- **RandomAccess** — добавляется поддержка произвольного доступа
- **Output** — итераторы, в которые можно писать

# Категории итераторов



# Операции над итераторами



2

# Операции над итераторами

- Разыменование — получение элемента, на который указывает итератор

# Операции над итераторами

- Разыменование — получение элемента, на который указывает итератор
- Инкремент — перемещение итератора вперед, для обращения к следующему элементу

# Операции над итераторами

- Разыменование — получение элемента, на который указывает итератор
- Инкремент — перемещение итератора вперед, для обращения к следующему элементу
- Декремент — перемещение итератора назад, для обращения к предыдущему элементу

# Операции над итераторами

- Разыменование — получение элемента, на который указывает итератор
- Инкремент — перемещение итератора вперед, для обращения к следующему элементу
- Декремент — перемещение итератора назад, для обращения к предыдущему элементу
- Проверка на равенство — два итератора равны тогда и только тогда, когда указывают на один и тот же элемент

# Операции над итераторами

- Разыменование — получение элемента, на который указывает итератор
- Инкремент — перемещение итератора вперед, для обращения к следующему элементу
- Декремент — перемещение итератора назад, для обращения к предыдущему элементу
- Проверка на равенство — два итератора равны тогда и только тогда, когда указывают на один и тот же элемент
- Проверка на неравенство — два итератора не равны, когда указывают на разные элементы



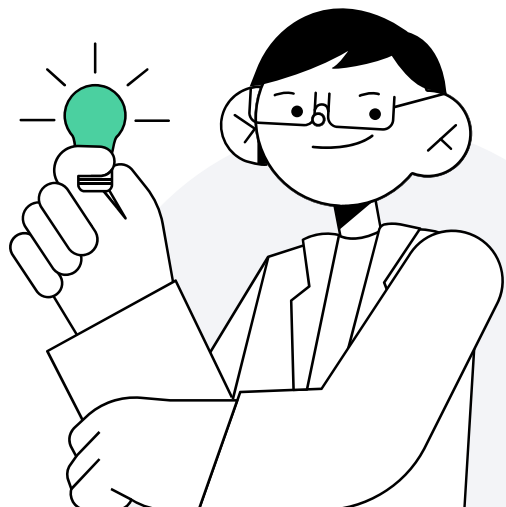
# Операции над итераторами

- Разыменование — получение элемента, на который указывает итератор
- Инкремент — перемещение итератора вперед, для обращения к следующему элементу
- Декремент — перемещение итератора назад, для обращения к предыдущему элементу
- Проверка на равенство — два итератора равны тогда и только тогда, когда указывают на один и тот же элемент
- Проверка на неравенство — два итератора не равны, когда указывают на разные элементы
- Операции сравнения (один итератор больше другого, если указывает на элемент, который ближе к концу)

# Итераторы контейнеров

Каждый контейнерный класс имеет 2 основных итератора:

- `container::iterator`
- `container::const_iterator`



# Итераторы контейнеров

Каждый контейнерный класс имеет 4 основных функции, возвращающих итераторы:

- `begin` — итератор на начальный элемент контейнера
- `end` — итератор на элемент, следующий за последним (сделано для удобства циклов)
- `cbegin` — константная версия итератора `begin`
- `cend` — константная версия итератора `end`

# Перерыв



# Примеры использования



2

# Обход контейнера

Рассмотрим обход контейнера с помощью итераторов:

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };  
std::vector<int>::iterator i = v.begin(); // итератор на начало  
while (i != v.end()) // пока не дошли до конца  
{  
    std::cout << *i << std::endl; // получаем элементы через итератор  
    i++; // перемещаемся вперед на один элемент  
}  
// OUTPUT: 0 1 2 3 4 5
```

# Обход контейнера. Константная версия

Если не предполагается изменение элементов, то лучше использовать константные версии итераторов:

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };
std::vector<int>::const_iterator i = v.cbegin(); // константный итератор на начало
while (i != v.cend()) // пока не дошли до конца
{
    std::cout << *i << std::endl; // получаем элементы через итератор
    i++; // перемещаемся вперед на один элемент
}
// OUTPUT: 0 1 2 3 4 5
```

# Обход контейнера с изменением элементов

Можно изменять элементы контейнера!

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };
std::vector<int>::iterator i = v.begin(); // итератор на начало
while (i != v.end()) // пока не дошли до конца
{
    *i = *i * 2; // меняем элемент
    std::cout << *i << std::endl; // получаем элементы через итератор
    i++; // перемещаемся вперед на один элемент
}
// OUTPUT: 0 2 4 6 8 10
```



# Обратный обход контейнера



**Вопрос:** что выведется в результате обхода контейнера?

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };  
std::vector<int>::reverse_iterator i = v.rbegin(); // итератор на начало  
while (i != v.rend()) // пока не дошли до конца  
{  
    *i = *i * 2; // меняем элемент  
    std::cout << *i << std::endl; // получаем элементы через итератор  
    i++; // перемещаемся вперед на один элемент  
}
```

# Обратный обход контейнера



**Вопрос:** что выведется в результате обхода контейнера?

**Ответ:** 10 8 6 4 2 0

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };
std::vector<int>::reverse_iterator i = v.rbegin(); // итератор на начало
while (i != v.rend()) // пока не дошли до конца
{
    *i = *i * 2; // меняем элемент
    std::cout << *i << std::endl; // получаем элементы через итератор
    i++; // перемещаемся вперед на один элемент
}
// OUTPUT: 10 8 6 4 2 0
```

# Обратный обход контейнера. Константная версия

Можно обойти контейнер, начиная с последнего элемента:

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };
std::vector<int>::const_reverse_iterator i = v.crbegin(); // итератор на начало
while (i != v.crend()) // пока не дошли до конца
{
    //*i = *i * 2; // так менять уже нельзя
    std::cout << *i << std::endl; // получаем элементы через итератор
    i++; // перемещаемся вперед на один элемент
}
// OUTPUT: 5 4 3 2 1 0
```

# Операции над итераторами в std::vector

Демонстрация работы с итераторами:

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };  
auto it = v.begin();  
auto it2 = it + 2; // 2  
std::cout << *it2;  
  
auto found = std::find(v.begin(), v.end(), 2);  
bool is_equal = it2 == found; // true  
  
it2++;  
is_equal = it2 == found; // false
```

# Операции над итераторами в std::set

Демонстрация работы с итераторами:

```
std::set<int> s = {0,1,2,3,4,5};

auto it = s.begin();
//auto it2 = it + 2; // не скомпилируется
auto it2 = it;
std::advance(it2, 2); // сдвиг на 2 элемента

auto found = std::find(s.begin(), s.end(), 2);
bool is_equal = it2 == found; // true

auto it2_next = std::next(it2); // 3
auto it2_prev = std::prev(it2); // 1
```

# Использование в алгоритмах и методах контейнеров



3

# Итераторы в методах контейнеров

У контейнеров есть основные методы для модификации содержимого:

- вставка
- удаление

```
std::vector <std::string > animals = { "dog", "lama", "cat", "tortoise" };  
auto it = find(animals.begin(), animals.end(), "lama"); // получаем позицию, на которой  
находится lama
```

```
animals.erase(it); // удаление элемента из контейнера по итератору it  
// "dog", "cat", "tortoise"
```

```
animals.insert(animals.begin(), "bear"); // вставка в начало  
// "bear", "dog", "cat", "tortoise"
```

```
auto it1 = find(animals.begin(), animals.end(), "cat");  
animals.erase(it1, animals.end()); // удалили все с cat и до конца  
// "bear", "dog"
```

# Итераторы в методах контейнеров

У insert есть несколько вариантов использования:

```
std::vector <std::string > animals = { "dog", "lama", "cat", "tortoise" };

animals.insert(animals.begin(), 2, "bear");
// "bear", "bear", "dog", "lama", "cat", "tortoise"

animals.insert(animals.begin(), { "horse", "fox" });
// "horse", "fox", "bear", "bear", "dog", "lama", "cat", "tortoise"
```



# Итераторы в алгоритмах. `std::remove_if`

Рассмотрим некоторые полезные алгоритмы, которые могут помочь вам с повседневными задачами. Например, удаление элементов из `std::vector`.

```
std::vector <std::string > animals = { "dog", "lama", "cat", "tortoise" };  
auto it = std::remove_if(animals.begin(), animals.end(),  
    [](const std::string& animal) {  
        return animal[1] == 'o';  
    })  
);  
animals.erase(it, animals.end());
```



**Задача:** удалить всех животных, у которых вторая буква — ‘o’.

**Вопрос:** Что же делает `remove_if`? Зачем нужно вызывать еще `erase`?

# Итераторы в алгоритмах. `std::remove_if`

```
std::vector <std::string > animals = { "dog", "lama", "cat", "tortoise" };  
auto it = std::remove_if(animals.begin(), animals.end(),  
    [](const std::string& animal) {  
        return animal[1] == 'o';  
    })  
);  
animals.erase(it, animals.end()); // "lama", "cat"
```



**Ответ:** `remove_if` сдвигает элементы внутри диапазона таким образом, что удаляемые элементы перезаписываются: все нужные элементы будут вначале.

`remove_if` возвращает итератор на новый конец нужного диапазона.

Поэтому в исходном векторе нам нужно удалить то, что осталось — вызвать функцию `erase`.

# Итераторы в алгоритмах. `std::unique`

```
std::vector <std::string > animals = { "dog", "lama", "cat", "cat", "dog" };  
auto it = std::unique(animals.begin(), animals.end());  
animals.erase(it, animals.end()); // "dog", "lama", "cat", "dog"
```

`std::unique` работает аналогичным образом: удаляет подряд идущие дубликаты .

Происходит сдвиг повторяющихся элементов в конец и возвращает итератор на конец нового диапазона.

# Итоги занятия

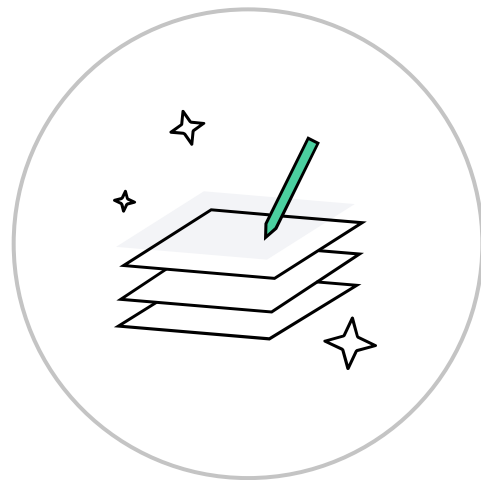
- 1 Узнали, что такое итераторы, какие они бывают и как использовать
- 2 Изучили некоторые сценарии использования итераторов



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



**Задавайте вопросы  
и пишите отзыв о лекции**

