

STL. Часть 1

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center



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

Амиран Мстоян

О спикере:

AI System Engineer (C++) в Huawei Moscow Research Center

Математик-алгоритмист лаборатории научного центра при МФТИ



Вспоминаем прошное занятие

Вопрос: что такое функтор?



Вспоминаем прошрое занятие

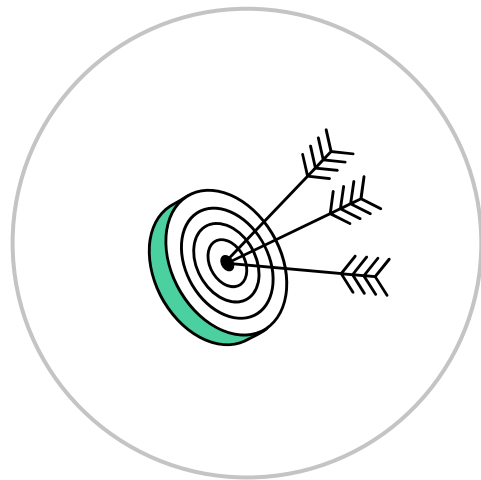
Вопрос: что такое функтор?

Ответ: это класс, в котором определен оператор ()



Цели занятия

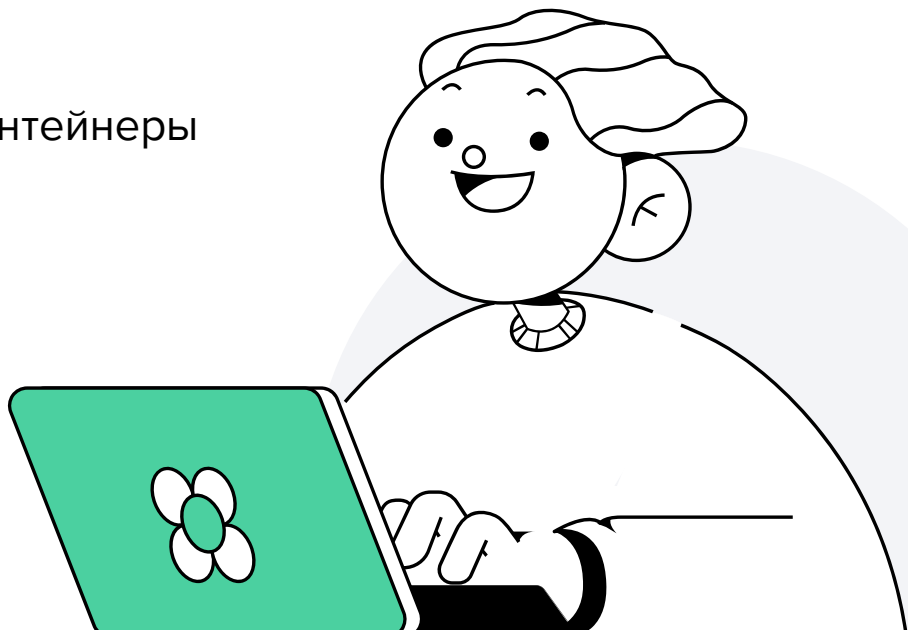
- Познакомимся с STL
- Узнаем основные контейнеры из STL



План занятия

- 1 STL
- 2 Последовательные контейнеры
- 3 Ассоциативные контейнеры
- 4 Неупорядоченные ассоциативные контейнеры
- 5 Адаптеры контейнеров
- 6 Домашнее задание

*Нажми на нужный раздел для перехода



STL



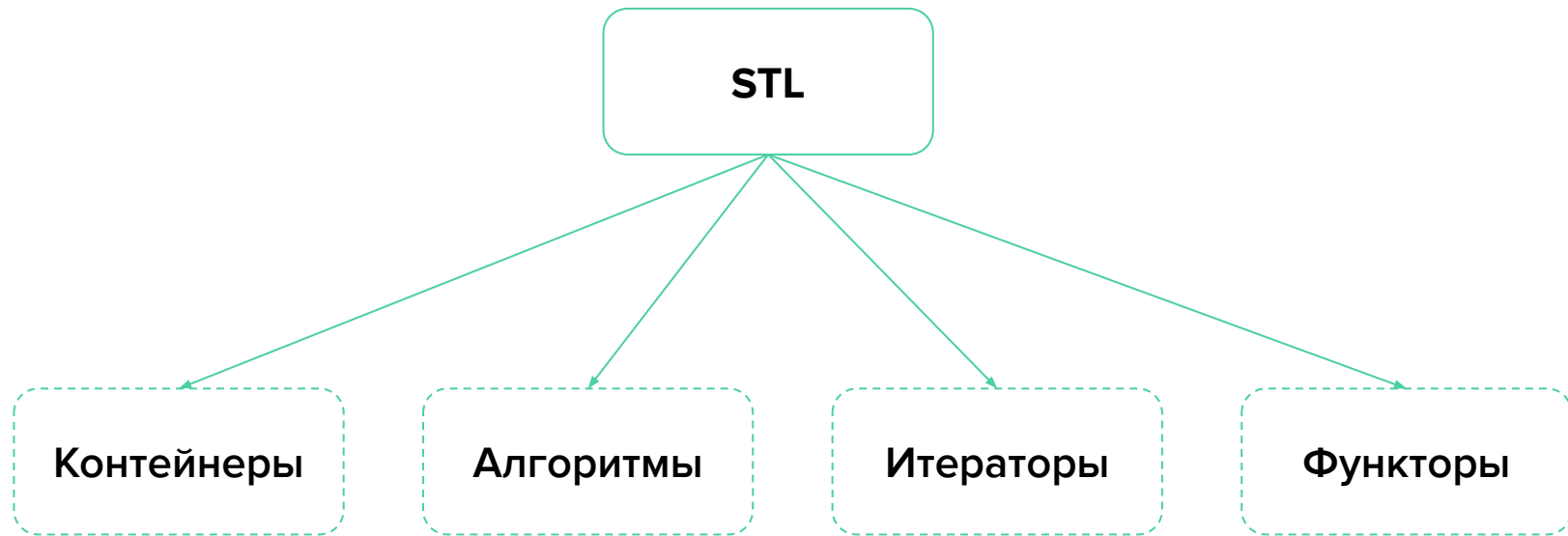
1



STL (standart template library, стандартная библиотека шаблонов) — набор обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++

Структура STL

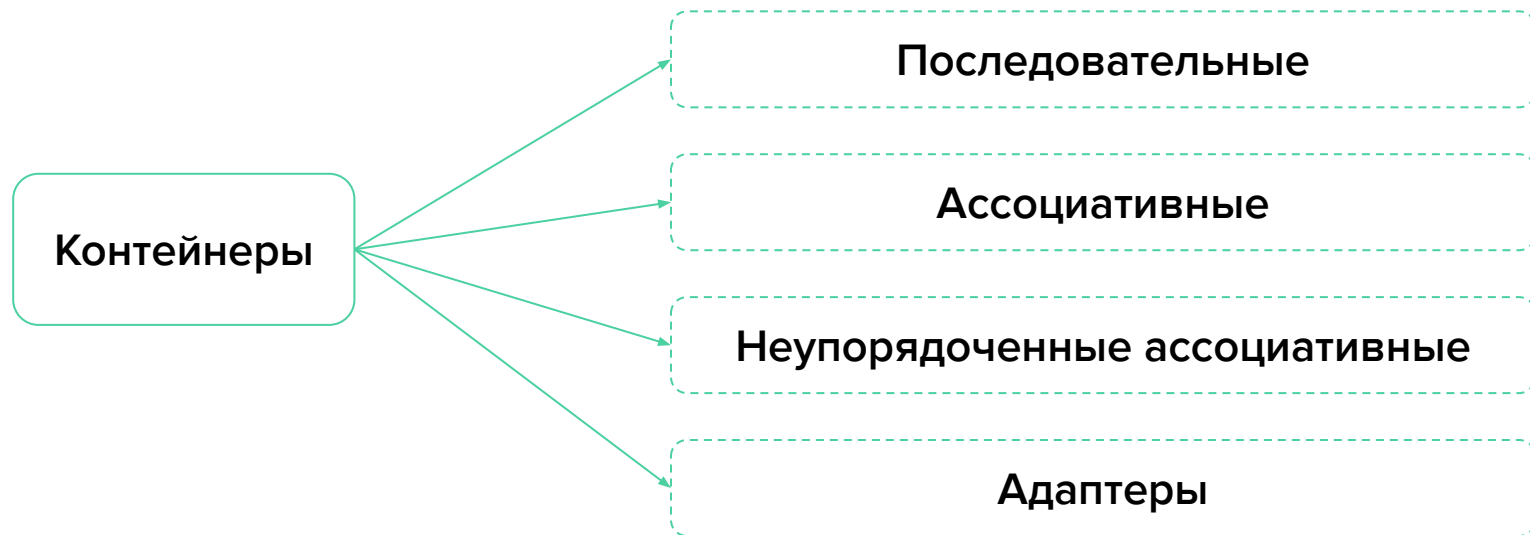
STL — очень обширная библиотека. Условно можно разделить ее на 4 блока



Контейнеры

Библиотека контейнеров является универсальной коллекцией шаблонов классов и алгоритмов, позволяющих программистам легко реализовывать общие структуры данных, такие как очереди, списки и стеки.

Существует 4 вида контейнеров:



Последовательные контейнеры



2

Последовательные контейнеры

Реализуют возможность последовательного доступа к элементам:

- 1 `std::array` — статический массив
- 2 `std::vector` — динамический массив
- 3 `std::deque` — двусторонняя очередь
- 4 `std::forward_list` — односвязный список
- 5 `std::list` — двусвязный список



std::array

Обыкновенный массив, **размер которого известен заранее**.

Представляет собой безопасную версию массивов в стиле C, так как сам управляет выделением и удалением памяти.

Элементы массива в памяти располагаются непрерывно друг за другом.



std::array

Используются следующие методы:

| Метод | Описание |
|-------------|---|
| at | Безопасный доступ к элементу массива (если индекс за границами массива, выбрасывает исключение) |
| оператор [] | Доступ к элементу массива (нет проверок индекса, но работает быстрее, чем at) |
| front | Первый элемент массива |
| back | Последний элемент массива |
| begin | Итератор на начало массива |
| end | Итератор на конец массива (точнее следующий за концом) |
| empty | Возвращает true, если пустой |
| size | Возвращает количество элементов |
| fill | Заполняет массив заданным значением |



std::array

Пример:

```
std::array<int, 3> arr1{ 1,4,5 }; // массив типа int с 3 элементами
std::array<char, 2> arr2 = { 'a','b' }; // массив типа char с 2 элементами

arr1[1] = 10; // arr1 = 1, 10, 5
arr2.fill('d'); // arr2 = d, d;

auto elem = arr1[5]; // неопределенное поведение
auto elem1 = arr1.at(5); // выбросит исключение

auto elem_front = arr1.front(); // 1
auto elem_back = arr1.back(); // 5

// поддержка range based for
for (const auto& elem : arr1)
    std::cout << elem << " ";
```




std::vector

Аналог `std::array` для случая, когда **размер неизвестен заранее**.

Поддерживает автоматическое управление памятью, элементы в памяти располагаются непрерывно друг за другом.



std::vector

Помимо методов, которые есть у std::array, добавляются следующие:

| Метод | Описание |
|---------------|--|
| push_back | Добавляет элемент в конец |
| pop_back | Удаляет элемент из конца |
| insert | Вставляет элементы |
| capacity | Возвращает количество элементов, которые могут храниться в выделенной в данный момент памяти |
| shrink_to_fit | уменьшает использование памяти за счёт освобождения неиспользуемой памяти |
| erase | Удаляет элемент по заданному индексу или по заданному диапазону |



std::vector

Пример:

```
std::vector<int> vec = {1, 5, 15, 4, 7, 8}
```

| | | | | | |
|---|---|----|---|---|---|
| 1 | 5 | 15 | 4 | 7 | 8 |
|---|---|----|---|---|---|

Вопрос на засыпку

Как бы вы реализовали функцию
`push_back` для вектора?

Напишите в чат



std::vector

Хочется заострить внимание на операции добавления нового элемента в вектор.

Так как мы не можем знать количество элементов заранее, то как узнать какое количество памяти нам необходимо выделить заранее?



`std::vector`

Для того, чтобы каждый раз при добавлении нового элемента не выделять новый блок памяти (а нужно скопировать еще и все предыдущие элементы в новую память!) решено было **выделять память с некоторым запасом**.

За этот запас и отвечает поле **capacity**.

Из-за такой особенности операции добавления нового элемента в вектор возникает некоторое неудобство: инвалидация ссылок.



std::vector

Основные функции для работы с std::vector

```
std::vector<int> v = { 7, 5, 16, 8 }; // size = 4, capacity = 4,
v.push_back(10); // v = {7, 5, 16, 8, 10} size = 5, capacity = 6
v.push_back(16); // v = {7, 5, 16, 8, 10, 16} size = 5, capacity = 6

v.pop_back(); // {7, 5, 16, 8, 10}

auto elem = v[10]; // неопределенное поведение
auto elem1 = v.at(10); // выбросит исключение

// поддержка range based for
for (const auto& elem : v)
    std::cout << elem << " ";
```



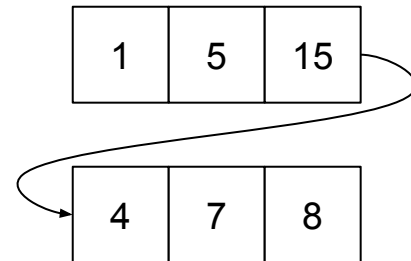
std::deque

Это двухсторонняя очередь: позволяет быстро вставлять и удалять элементы как в начале, так и в конце.

В отличие от std::vector, элементы двухсторонней очереди не хранятся непрерывно: типичные реализации используют последовательность отдельно выделенных массивов фиксированного размера.

Из-за такого способа хранения расширение двухсторонней очереди проходит дешевле (не нужно копировать все элементы, а можно просто добавить новый блок), но индексация может быть чуть дольше.

```
std::deque<int> vec = {1, 5, 15, 4, 7, 8}
```





std::deque

Используются следующие методы:

| Метод | Описание |
|------------|---|
| push_back | Добавляет элемент в конец |
| pop_back | Удаляет элемент из конца |
| push_front | Добавляет элемент в начало |
| pop_front | Удаляет элемент из начала |
| insert | Вставляет элементы |
| erase | Удаляет элемент по заданному индексу или по заданному диапазону |



std::deque

Описание

```
std::deque<int> d = { 7, 5, 16, 8 };  
d.push_back(10); // v = {7, 5, 16, 8, 10}  
d.push_front(16); // v = {16, 7, 5, 16, 8, 10}  
  
d.pop_back(); // {16, 7, 5, 16, 8}  
d.pop_front(); // {7, 5, 16, 8}  
  
auto elem = d[10]; // неопределенное поведение  
auto elem1 = d.at(10); // выбросит исключение  
  
// поддержка range based for  
for (const auto& elem : d)  
    std::cout << elem << " ";
```

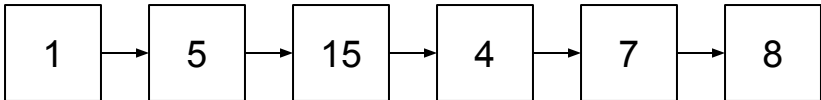


std::forward_list

Односвязный список — поддерживается **итерация только в одном направлении**.

Этот контейнер предоставляет быструю операцию удаления и вставки элемента в произвольное место. Однако, нет быстрого доступа по индексу, так как каждому элементу контейнера соответствует свой блок памяти.

```
std::forward_list<int> vec = {1, 5, 15, 4, 7, 8}
```





std::forward_list

Используются следующие методы:

| Метод | Описание |
|--------------|---|
| push_front | Добавляет элемент в начало |
| pop_front | Удаляет элемент из начала |
| insert_after | Вставляет элементы после заданного |
| erase_after | Удаляет элементы после заданного |
| remove | Удаляет элементы, удовлетворяющие определённным критериям |
| unique | Удаляются последовательно повторяющиеся элементы |
| sort | Сортирует элементы |



std::forward_list

Описание

```
std::forward_list<int> list = { 1,5,4,2,3 };  
list.sort(); // 1 2 3 4 5  
  
list.insert_after(list.before_begin(), { 6, 7, 8 }); // 6 7 8 1 2 3 4 5  
  
list.insert_after(  
    std::find(list.begin(), list.end(), 2),  
    5); // 6 7 8 1 2 5 3 4 5  
  
// поддержка range based for  
for (const auto& elem : list)  
    std::cout << elem << " ";
```

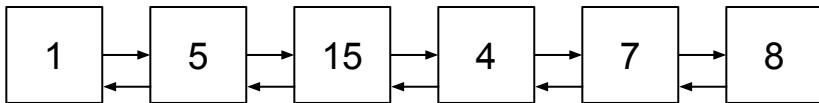


std::list

Двусвязный список — в отличие от `forward_list` **обеспечивает возможность двунаправленного итерирования**.

Этот контейнер предоставляет быструю операцию удаления и вставки элемента в произвольное место. Однако, нет быстрого доступа по индексу, так как каждому элементу контейнера соответствует свой блок памяти.

```
std::forward_list<int> vec = {1, 5, 15, 4, 7, 8}
```





std::list

Используются следующие методы:

| Метод | Описание |
|------------|--|
| push_front | Добавляет элемент в начало |
| pop_front | Удаляет элемент из начала |
| push_back | Добавляет элемент в конец |
| pop_back | Удаляет элемент в конец |
| insert | Вставляет элементы на заданное место |
| erase | Удаляет элементы |
| remove | Удаляет элементы, удовлетворяющие определённому критерию |
| unique | Удаляются последовательно повторяющиеся элементы |
| sort | Сортирует элементы |



std::list

Примеры работы с контейнером std::list

```
std::list<int> list = { 1,5,4,2,3 };  
list.sort(); // 1 2 3 4 5  
  
list.insert(list.begin(), { 6, 7, 8 }); // 6 7 8 1 2 3 4 5  
  
list.insert(  
    std::find(list.begin(), list.end(), 2),  
    5); // 6 7 8 1 2 5 3 4 5  
  
// поддержка range based for  
for (const auto& elem : list)  
    std::cout << elem << " ";
```


Ассоциативные контейнеры



3

Ассоциативные контейнеры

Реализуют упорядоченные структуры данных с возможностью быстрого поиска ($O(\log N)$).

По умолчанию все элементы сортируются в порядке возрастания ключей.

- 1 `std::set` — множество с уникальными объектами
- 2 `std::map` — словарь с уникальными ключами
- 3 `std::multiset` — множество с поддержкой дубликатов
- 4 `std::multimap` — словарь с поддержкой дубликатов



std::set

Множество с отсортированными по возрастанию элементами.

У типа элементов, которые будут храниться в контейнере, должен быть определён оператор `<`.

std::multiset устроен также, но элементы в нем могут быть не уникальными.



std::set

Используются следующие методы:

| Метод | Описание |
|-------------|---|
| empty | Проверяет, есть ли элементы |
| clear | Очищает контейнер |
| insert | Вставляет элементы |
| erase | Удаляет элементы |
| count | Возвращает кол-во элементов, соответствующих заданному |
| find | Находит заданный элемент |
| lower_bound | Находит первый элемент не меньший (равный или больший), чем заданное значение |
| upper_bound | Находит первый элемент больший, чем заданное значение |



std::set

Описание

```
std::set<int> my_set;  
my_set.insert(10); // 10  
my_set.insert(5); // 5 10  
my_set.insert(6); // 5 6 10  
my_set.insert(6); // 5 6 10  
  
auto elem_l = my_set.lower_bound(7); // 10  
  
// поддержка range based for  
for (const auto& elem : my_set)  
    std::cout << elem << " ";
```



std::map

Контейнер, который содержит пары ключ-значение с уникальными ключами.

Ключи сортируются с помощью функции сравнения.

std::multimap устроен также, но элементы в нем могут быть не уникальными.



std::map

Используются следующие методы:

| Метод | Описание |
|-------------|---|
| empty | Проверяет, есть ли элементы |
| clear | Очищает контейнер |
| insert | Вставляет элементы |
| erase | Удаляет элементы |
| count | Возвращает кол-во элементов, соответствующих заданному |
| find | Находит заданный элемент |
| lower_bound | Находит первый элемент не меньший (равный или больший), чем заданное значение |
| upper_bound | Находит первый элемент больший, чем заданное значение |



std::map

Описание

```
std::map<std::string, int> m{
    {"CPU", 10}, {"GPU", 15}, {"RAM", 20}
};

m["CPU"] = 25; // {"CPU", 25}, {"GPU", 15}, {"RAM", 20}

// использование operator[] с несуществующим ключом всегда выполняет вставку
m["SSD"] = 30; // {"CPU", 25}, {"GPU", 15}, {"RAM", 20}, {"SSD", 30}

m.erase("GPU"); // {"CPU", 25}, {"RAM", 20}, {"SSD", 30}

// поддержка range based for
for (const auto& elem : m)
    std::cout << elem.first << ": " << elem.second << std::endl;
```


Перерыв



Неупорядоченные ассоциативные контейнеры



4

Неупорядоченные ассоциативные контейнеры

Реализуют неупорядоченные структуры данных с возможностью быстрого поиска ($O(1)$ или $O(N)$).

Имеют аналогичные функции с упорядоченными ассоциативными контейнерами.

- 1 `std::unordered_set` — множество с уникальными объектами
- 2 `std::unordered_map` — словарь с уникальными ключами
- 3 `std::unordered_multiset` — множество с поддержкой дубликатов
- 4 `std::unordered_multimap` — словарь с поддержкой дубликатов

Адаптеры контейнеров



5

Адаптеры контейнеров

Адаптеры контейнеров предоставляют различные интерфейсы для последовательных контейнеров.

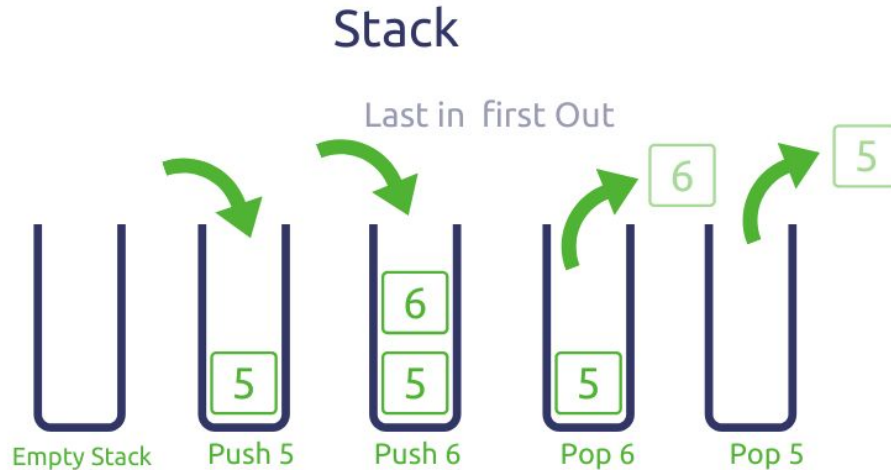
- 1 `std::stack` — стек
- 2 `std::queue` — очередь
- 3 `std::priority_queue` — очередь с приоритетами

1

std::stack

Структура данных, организованная по принципу LIFO.

LIFO (Last in first, first out) — последний добавленный элемент в контейнер, первый будет удален из контейнера. Стек добавляет новые элементы в конец контейнера (вершина стека) и удаляет с конца.





std::stack

Используются следующие методы:

| Метод | Описание |
|-------|--|
| top | Доступ к элементу на вершине стека |
| empty | Проверяет, пуст ли контейнер |
| size | Возвращает кол-во элементов в контейнере |
| push | Добавляет элемент на вершину |
| pop | Удаляет элемент с вершины |



std::stack

Описание

```
std::stack<int> st;  
st.push(5);  
st.push(15); // 5 15  
auto elem = st.top(); //15  
  
st.push(2); // 5 15 2  
st.pop(); // 5 15  
elem = st.top(); // 15
```


2

`std::queue`

Структура данных, организованная по принципу FIFO (First in first, first out): первый добавленный элемент в контейнер, первым будет удален из контейнера.

Очередь добавляет новый элемент в конец и удаляет элементы с начала.





std::queue

Используются следующие методы:

| Метод | Описание |
|-------|--|
| back | Доступ к последнему элементу |
| front | Доступ к первому элементу |
| empty | Проверяет, пуст ли контейнер |
| size | Возвращает кол-во элементов в контейнере |
| push | Добавляет элемент в конец |
| pop | Удаляет первый элемент |



std::queue

Описание

```
std::queue<int> st;  
st.push(5); // 5  
st.push(15); // 5 15  
st.push(7); // 5 15 7  
auto elem = st.back(); // 7  
elem = st.front(); // 5  
st.pop(); // 15 7
```

3

`std::priority_queue`

Очередь с приоритетом. Обеспечивает константное время поиска самого большого (по умолчанию) элемента.

Это обеспечивается за счет того, что элементы сортируются при вставке и извлечении.



Итоги



Итоги занятия

- 1 Познакомились со структурой STL
- 2 Изучили контейнеры из STL: последовательные, ассоциативные, неупорядоченные ассоциативные, контейнеры-адаптеры



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



Дополнительные материалы

- [Документация](#) по контейнерам



Задавайте вопросы и пишите отзыв о лекции

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center

