Углублённое программирование на C++

Standard Template Library
Эпизод I
Контейнеры наносят ответный удар

Кухтичев Антон

education

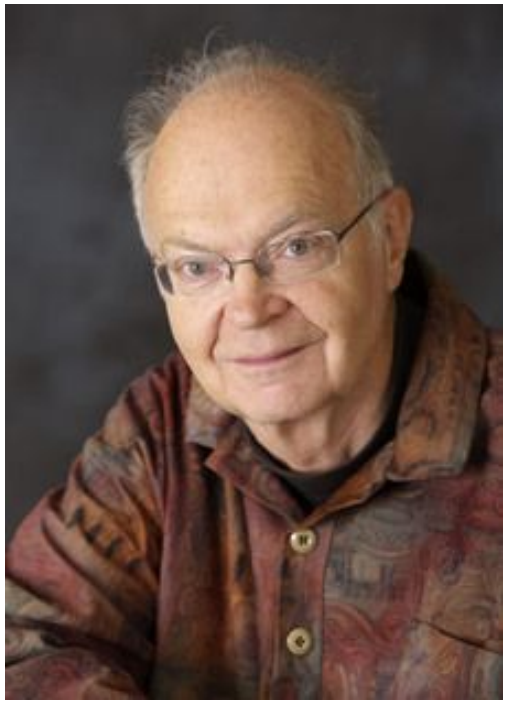
02 июля 2024 года

Содержание занятия

- Квиз по исключениям
- Стандартная библиотека C++
- Вспомогательные классы
- Библиотека шаблонов STL (Standard Template Library)

Угадай программиста по фотографии

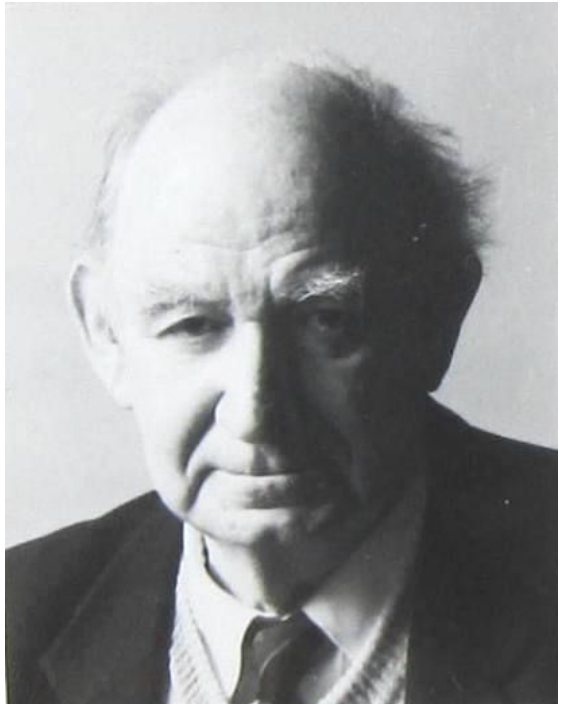
Кто это?



Дональд Эрвин Кнут

Кент Бэк

Джеффри Ульман



Е.М. Ландис

Квиз по исключениям

Стандартная библиотека C++

Стандартная библиотека C++

1. Ввод-вывод
2. Многопоточность
3. Регулярные выражения
4. Библиотека C
5. Библиотека шаблонов STL
6. Прочее (дата и время, обработка ошибок, поддержка локализации и т.д.)

Документация: https://en.cppreference.com/w/

std::ifstream

```
std::ifstream file("/tmp/file.txt");
if (!file)
{
    std::cout << "can't open file" ;
    return;
}

while (file.good())
{
    std::string s;
    file >> s;
}
```

std::ifstream

```
const int size = 1024;
char buf[size];

std::ifstream file("/tmp/file.data", std::ios::binary);
file.read(buf, size);
const auto read = file.gcount();
```

std::ofstream

Запись в файл.

```
std::ofstream file("/tmp/file.txt");
if (!file)
{
    std::cout << "can't open file" ;
    return;
}

file << "abc" << 123;
```

std::ofstream

```
const int size = 1024;
char buf[size];

std::ofstream file("/tmp/file.data", std::ios::binary);
file.write(buf, size);
```

Вспомогательные классы

std::optional

Шаблон класса, который содержит значение, которое может присутствовать или не присутствовать.

- `has_value()` — проверка, содержит ли переменная объект;
- `value()` — возвращает ссылку на объект, содержащийся в переменной;
- `operator=` — присвоить объект;
- `reset()` — уничтожает объект, содержащийся в переменной;
- `std::nullopt` — константа, обозначающая неинициализированное состояние.

std::optional

В инструкциях, где ожидается `bool` (`if`, `while`, `for`, `?:`) `std::optional` возвращает `true`, если в нём содержится объект, и `false` в противном случае.

- `if (opt)` эквивалентно `if (opt.has_value())`
- Объект может иметь значение `false`;

```
std::optional<bool> opt1 = false;
if (opt1) {
    // выполнится условие
}
std::optional<void*> opt2 = nullptr;
if (opt2) {
    // выполнится условие
}
```

tribool

Тип, подобный `bool`, который поддерживает три состояния, а не два: истина, ложь и неопределённость.

- Неявно преобразуется в `bool`. Если `tribool` имеет значение `true`, он преобразуется в `true`, в противном случае - в `false`;

```
#include <boost/logic/tribool.hpp>
using boost::logic::indeterminate;
boost::logic::tribool t = true, f = false, i = indeterminate;
```

std::pair

Тип позволяющий упаковать два значения в один объект.

```
#include <utility>

auto p1 = std::pair<int, double>(1, 2.0);
auto p2 = std::make_pair(1, 2.0);

auto x = p1.first; // int == 1
auto y = p1.second; // double == 2
```

`pair` имеет операторы сравнения позволяющие сделать лексикографическое сравнение элементов.

std::tuple

Тип позволяющий упаковать несколько значений в один объект.

```
#include <tuple>

auto t = std::make_tuple(1, 2.0, "abc");
int a = std::get<0>(t);
double b = std::get<1>(t);
std::string c = std::get<2>(t);
```

```
auto [a, b, c] = t;
```

Соответствие типов проверяется на этапе компиляции.

Как и `pair` имеет лексикографические операторы сравнения.

std::tie

`tie`, как и `make_tuple` создает `tuple`, но не объектов, а ссылок на них.

Использование tie для написания операторов сравнения

```
struct MyClass
{
    int x_;
    std::string y_;
    double z_;

    bool operator<(const MyClass& o) const
    {
        return std::tie(x_, y_, z_) < std::tie(o.x_, o.y_, o.z_);
    }
};
```

Использование tie для написания операторов сравнения

```
bool operator<(const MyClass& o) const
{
    if (x_ != o.x_)
        return x_ < o.x_;
    if (y_ != o.y_)
        return y_ < o.y_;
    return z_ < o.z_;
}
```

Библиотека шаблонов STL (Standard Template Library)

Библиотека шаблонов STL (Standard Template Library)

1. Контейнеры (containers) – хранение набора объектов в памяти
2. Итераторы (iterators) – средства для доступа к источнику данных (контейнер, поток)
3. Алгоритмы (algorithms) – типовые операции с данными
4. Адаптеры (adaptors) – обеспечение требуемого интерфейса
5. Функциональные объекты (functors) – функция как объект для использования другими компонентами

O большое

«O» большое – математическое обозначение для сравнения асимптотического поведения алгоритма.

Фраза «сложность алгоритма есть `O(f(n))`» означает, что с ростом параметра `n` время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на `f(n)`.

O большое

Типичные значения:

1. Время выполнения константно: $O(1)$
2. Линейное время: $O(n)$
3. Логарифмическое время: $O(\log n)$
4. Время выполнения «n логарифмов n»: $O(n \log n)$
5. Квадратичное время: $O(n^2)$

Контейнеры

1. Последовательные (Sequence containers)
2. Ассоциативные (Associative containers)
3. Неупорядоченные ассоциативные (Unordered associative containers)
4. Контейнеры-адаптеры (Container adaptors)

Последовательные контейнеры

std::array

```cpp
#include <array>

template <class T, size_t N>
class array
{
    T data_[N];
    size_t size_;
public:
    using size_type = size_t;
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
};
```

std::array

- `constexpr size_type size() const noexcept;`
- `constexpr bool empty() const noexcept;`
- `reference at(size_type pos);`
- `constexpr const_reference at(size_type pos) const;`
- `reference operator[](size_type pos);`
- `constexpr const_reference operator[](size_type pos) const;`
- `reference front();`
- `constexpr const_reference front() const;`
- `reference back();`
- `constexpr const_reference back() const;`
- `T* data() noexcept;`
- `const T* data() const noexcept;`

std::array

Пример:

```
std::array<int, 5> a = { 1, 2, 3, 4, 5 };
auto x = a[2];
a[2] = x * 2;
```

Вставка	Удаление	Поиск	Доступ
-	-	O(n)	O(1)

std::initializer_list

```
template <class T>
class initializer_list
{
public:
    size_type size() const noexcept;
    const T* begin() const noexcept;
    const T* end() const noexcept;
};
```

std::initializer_list

```
Array<int, 3> a = { 1, 2, 3 };

template <class T, size_t N>
class Array
{
public:
    Array(std::initializer_list<T> init)
    {
        size_t i = 0;
        auto current = init.begin();
        const auto end = init.end();
        while (current != end)
        {
            data_[i++] = *current++;
        }
    }
};
```

Аллокаторы

Назначение аллокатора - выделять и освобождать память.

`malloc` и `new` - аллокаторы

```
std::allocator<int> a1;
int* a = a1.allocate(1);
a1.construct(a, 7);
std::cout << a[0] << '\n';
a1.deallocate(a, 1);
```

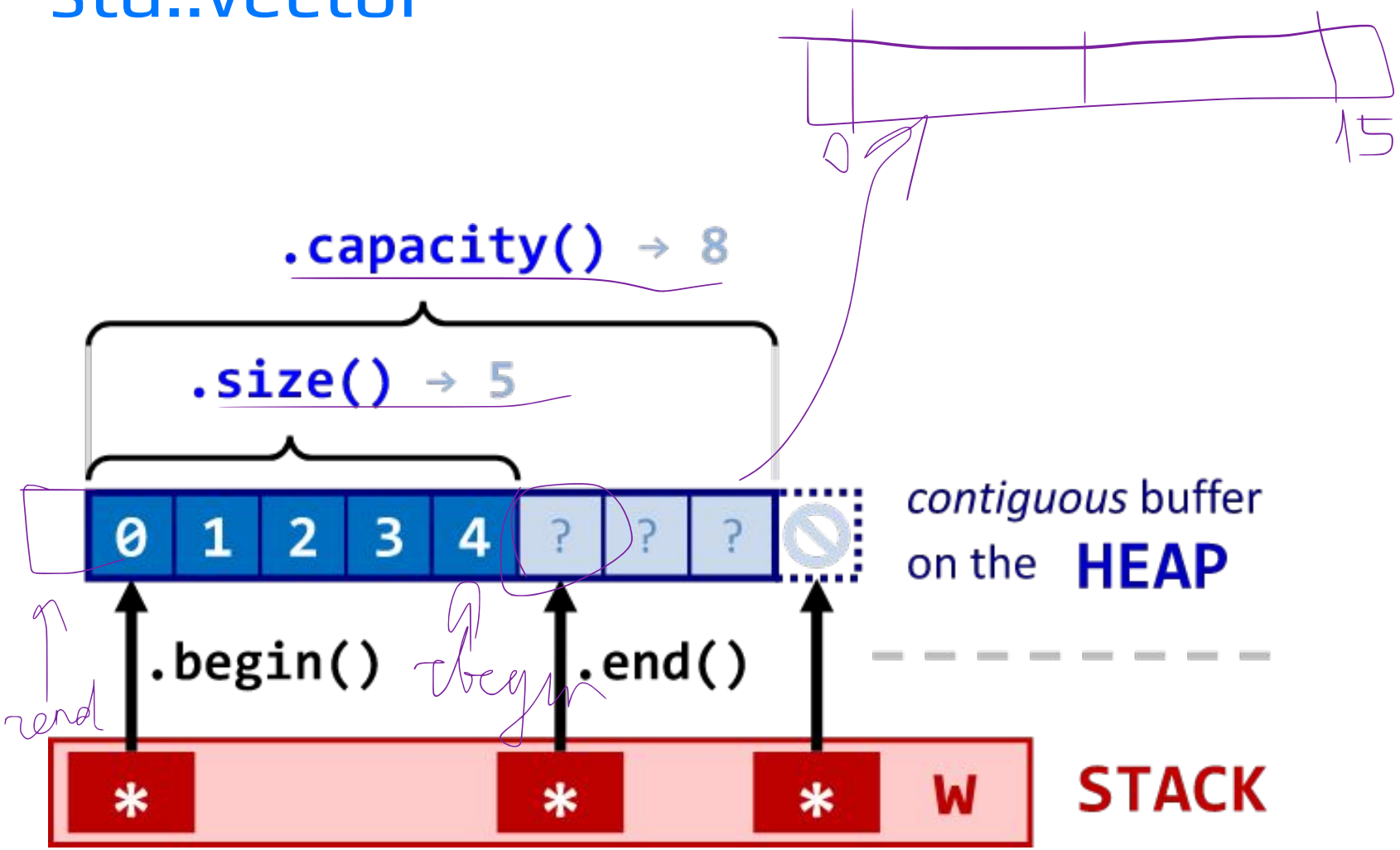
Аллокаторы

```
template<class T>
class allocator
{
public:
    using value_type = T;
    using pointer = T*;
    using size_type = size_t;

    pointer allocate(size_type count);
    void deallocate(pointer ptr, size_type count);

    // deprecated in C++17, removed in C++20
    size_t max_size() const noexcept;
};
```

std::vector


0
15
.capacity() → 8
.size() → 5
0 1 2 3 4 ? ? ?
contiguous buffer
on the HEAP
.begin()
.end()
rend
begin
*
*
*
W
STACK

std::vector

```
template<class T, class Alloc = std::allocator<T>>
class vector
{
public:
    using size_type = size_t;
    using value_type = T;
    using reference = T&;
    using const_reference = const T&;
    using allocator_type = Alloc;
```

std::vector

```
explicit vector(size_type count);
vector(size_type count, const value_type& defaultValue);
vector(initializer_list<value_type> init);

iterator begin() noexcept; // аналогично end()
reverse_iterator rbegin() noexcept; // rend()
const_iterator cbegin() const noexcept; // cend()
const_reverse_iterator crbegin() const noexcept;// crend()
```

std::vector

```
void reserve(size_type count); // Выделяет память
size_type capacity() const noexcept;
void push_back(const value_type& value);

template<class... VT>
void emplace_back(VT&&... values);

iterator insert(const_iterator where, const T& value);
```

std::vector

Пример:

```
class A
{
    A(int, int) {}
    A(A&&) {}
};

A a(1, 2);

vec.push_back(std::move(a));
vec.emplace(1, 2);
```

Вставка	**Удаление**	**Поиск**	**Доступ**
В конце O(1) в среднем или O(n)	В конце O(1) или O(n)	O(n) В отсортированном O(log n)	O(1)

Трюки с вектором

1. Если порядок элементов не важен, то меняем удаляемый элемент с последним местами и удаляем последний (pop_back).
2. Изменение размера вектора перед вставкой

```
const auto size = file.size();
std::vector<char> data(size);
for (size_t i = 0; i < size; ++i)
    data[i] = file.read();
```

1. Очистка вектора

std::deque

Интерфейс повторяет интерфейс `std::vector`, отличие в размещении в памяти - `std::vector` хранит данные в одном непрерывном куске памяти, `std::deque` хранит данные в связанных блоках по n элементов.

Вставка	**Удаление**	**Поиск**	**Доступ**
В конце O(1) или O(n)	В конце O(1) или O(n)	O(n) В отсортированном O(log n)	O(1)

std::forward_list

Связный список, элементы которого хранятся в произвольных участках памяти.

Итератор списка не поддерживает произвольный доступ, следовательно алгоритмы STL, которые требуют random access iterator работать со списком не будут, например, `std::sort`.

Вставка	**Удаление**	**Поиск**	**Доступ**
O(1)	O(1)	O(n)	O(n)

std::list

Отличие от односвязного списка - возможность перемещаться в обратном направлении.

```
template <class T>
class Node
{
    T value_;
    Node<T>* prev_;
    Node<T>* next_;
};
```

Вставка	**Удаление**	**Поиск**	**Доступ**
O(1)	O(1)	O(n)	O(n)

Ассоциативные контейнеры

Контейнер позволяющий хранить пары вида (ключ, значение) и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Элементы **отсортированы** по ключу:

1. `set<Key, Compare, Allocator>`
2. `map<Key, T, Compare, Allocator>`
3. `multiset<Key, Compare, Allocator>`
4. `multimap<Key, T, Compare, Allocator>`

Ассоциативные контейнеры

Элементы **не отсортированы** по ключу:

1. `unordered_set<Key, Hash, KeyEqual, Allocator>`
2. `unordered_map<Key, T, Hash, KeyEqual, Allocator>`
3. `unordered_multiset<Key, Hash, KeyEqual, Allocator>`
4. `unordered_multimap<Key, T, Hash, KeyEqual, Allocator>`

В `unordered_set` и `unordered_map` ключи уникальны, в multi версиях контейнеров допускаются наличие значений с одинаковым ключом.

Вставка	**Удаление**	**Поиск**	**Доступ**
O(1) или O(n)	O(1) или O(	O(1)	O(1)

Ассоциативные контейнеры

```
#include <unordered_map>

std::unordered_map<std::string, size_t> frequencyDictionary;
std::string word;
while (getWord(word))
{
    auto it = frequencyDictionary.find(word);
    if (it == frequencyDictionary.end())
        frequencyDictionary[word] = 1;
    else
        it->second++;
}
```

Контейнеры-адаптеры

Контейнеры-адаптеры

Являются обертками над другими контейнерами и предоставляют нужный интерфейс.

1. `stack<T, Container = std::deque<T>>`
2. `queue<T, Container>`
3. `priority_queue<T, Container, Compare>`

std::stack

Реализует интерфейс стека - положить значение в стек, извлечь значение из стека, последний пришел первый вышел (LIFO).

```
#include <stack>

std::stack<int> s;
s.push(3);
s.push(5);
int x = s.top(); // 5
s.pop();
int y = s.top(); // 3
```

std::queue

Реализует интерфейс очереди - положить значение в стек, извлечь первое значение из стека, первый пришел первый вышел (FIFO).

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

void push(const value_type& value);
void push(value_type&& value);
reference front();
const_reference front() const;
void pop();
```

std::priority_queue

Отличие от queue - за O(1) можно извлечь элемент наиболее полно удовлетворяющий условию.

```
#include <queue>

template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

Code time

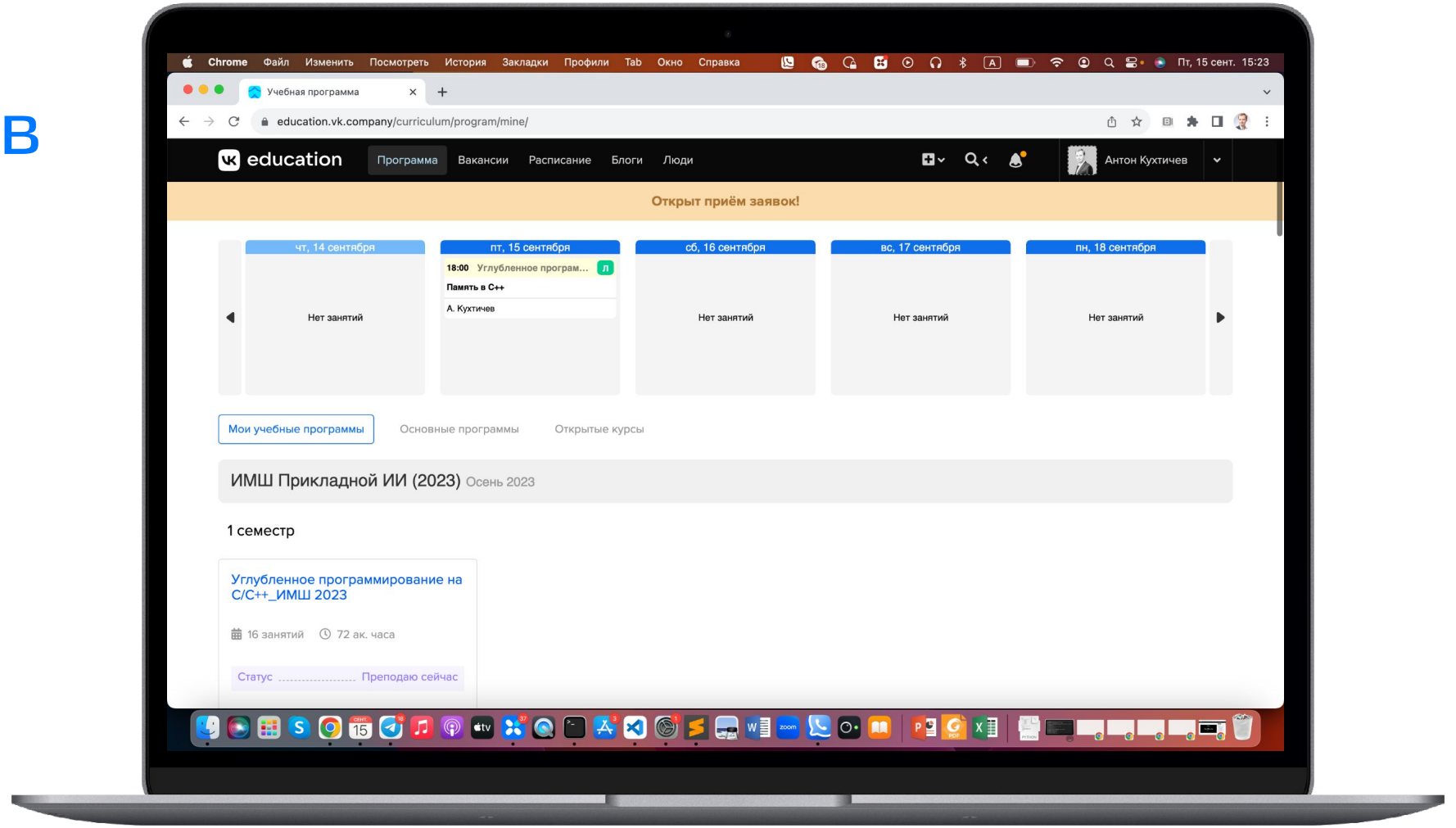
- Хотим получить TOP-k наибольших элементов в массиве!

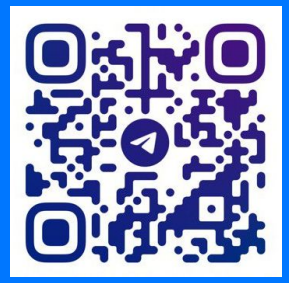
Полезная литература в помощь

- Документация стандартной библиотеки

Напоминание оставить отзыв

Это правда важно





Спасибо
за внимание!