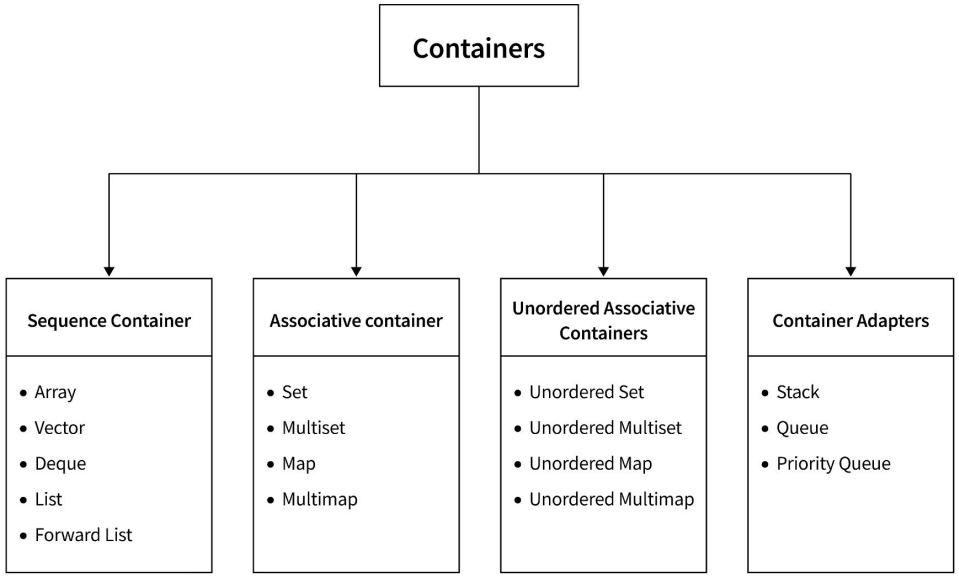
!STL, STD::;

Что такое STL, std?

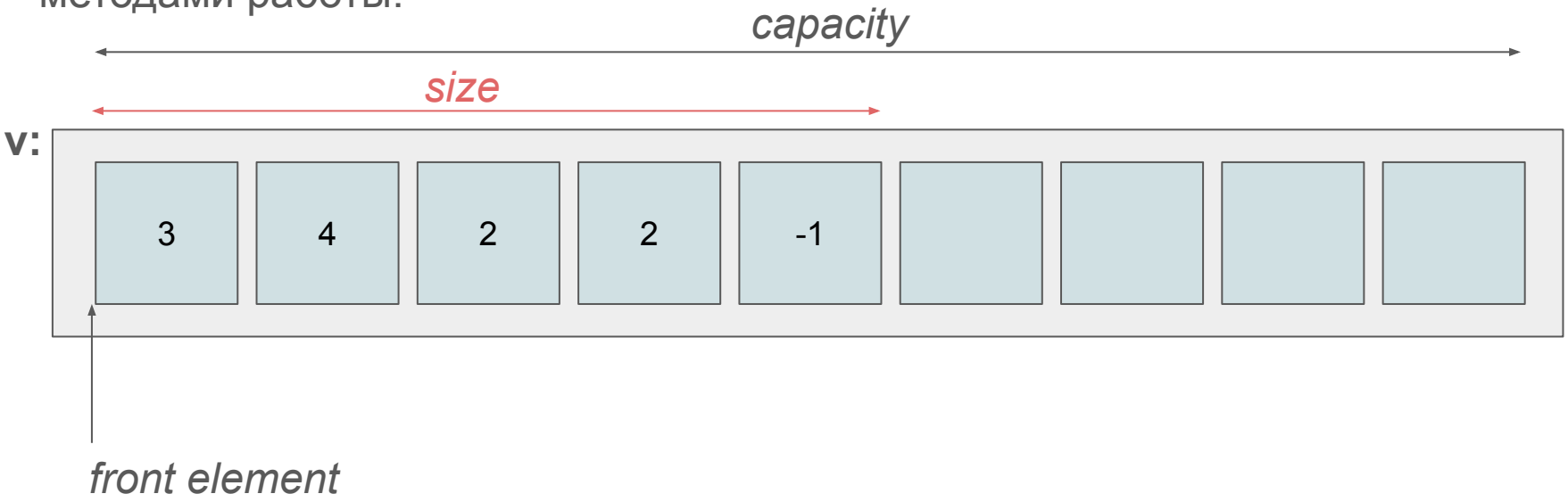
1. **STL (Standard Template Library):** Это набор шаблонов классов и функций в стандартной библиотеке C++, предназначенный для обработки данных и выполнения общих задач программирования.
2. **std:** Это пространство имен в стандартной библиотеке C++, где располагаются все классы, функции и объекты из STL и других частей стандартной библиотеки.



vector (вектор)

Что такое vector?

Вектор - это шаблонный динамический массив данных с определенными методами работы.



методы работы над вектором.

1. **push_back():** Добавляет элемент в конец вектора.
2. **pop_back():** Удаляет последний элемент из вектора.
3. **size():** Возвращает количество элементов в векторе.
4. **empty():** Проверяет, пуст ли вектор.
5. **clear():** Очищает содержимое вектора.
6. **resize():** Изменяет размер вектора.
7. **reserve():** Выделяет память для указанного количества элементов без изменения их количества.
8. **operator[]:** Доступ к элементу вектора по индексу.
9. **at():** Доступ к элементу вектора с проверкой диапазона.
10. **front():** Возвращает первый элемент вектора.
11. **back():** Возвращает последний элемент вектора.
12. **erase():** Удаляет один или несколько элементов из вектора по указанной позиции или диапазону.
13. **insert():** Вставляет элемент(ы) в указанную позицию вектора.
14. **assign():** Заменяет содержимое вектора указанным количеством элементов или значениями.
15. **emplace_back():** Создает и добавляет элемент в конец вектора на месте без копирования или перемещения.
16. **emplace():** Вставляет элемент в указанную позицию вектора на месте без копирования или перемещения.
17. **begin():** Возвращает итератор, указывающий на начало вектора.
18. **end():** Возвращает итератор, указывающий на конец вектора.
19. **rbegin():** Возвращает обратный итератор, указывающий на последний элемент вектора.
20. **rend():** Возвращает обратный итератор, указывающий на элемент перед началом вектора.
21. **capacity():** Возвращает текущую вместимость (количество элементов, которые могут быть сохранены без перевыделения памяти) вектора.
22. **data():** Возвращает указатель на внутренний массив вектора.
23. **max_size():** Возвращает максимальное количество элементов, которое вектор может содержать.

базовый класс vector

```
class Std {
public:
    template<typename T>
    class vector {
    private:
        T* data;
        size_t capacity;
        size_t length;
    public:
        vector() : capacity(2), length(0), data(new T[capacity]) {}
        ~vector() {delete[] data;}
```

СОЗДАНИЕ ВЕКТОРА

```
Std::vector<int> v{};
```

push_back

1. Проверить, достигла ли длина вектора его текущей вместимости.
   `if (length == capacity) {..}`
2. Если вектор заполнен, увеличить его вместимость в два раза.
3. Скопировать все существующие элементы в новый массив с увеличенной вместимостью.
4. Добавить новый элемент в конец вектора.
5. Увеличить длину вектора.

```
void push_back(T value) {...}
```

pop_back

1. Уменьшить длину вектора на 1.
2. Если длина вектора стала меньше его вместимости раз в два, уменьшить вместимость вектора вдвое.
3. Если вместимость вектора стала меньше определенного порога (например, четверти от предыдущей вместимости), перевыделить память с уменьшенной вместимостью.

```
void pop_back() {
    if (length > 0) --length;
}
```

size

Для получения размера (количества элементов) вектора просто возвращаем значение переменной length. Вернуть значение переменной length.

```
[[nodiscard]] size_t size() const {return length;}
```

empty/clear/resize

empty():

1. Проверить, равна ли длина вектора нулю.
2. Вернуть true, если длина равна нулю, иначе вернуть false.

```
bool empty() const {
    return length == 0;
}
```

clear():

1. Установить длину вектора в ноль.

```
void clear() {
    length = 0;
}
```

resize():

1. Установить новую длину вектора.
2. Если новая длина больше текущей, заполнить новые элементы значением по умолчанию.
3. Если новая длина меньше текущей, обрезать вектор.

```
void resize(size_t new_size) {
    if (new_size > length) {
        if (new_size > capacity) {
            reserve(new_size);
        }
        for (...) {
            // TODO
        }
    } else {
        length = new_size;
    }
}
```

reserve

1. Проверить, нужно ли увеличить вместимость вектора.
2. Если необходимо, выделить память для новой вместимости и скопировать существующие элементы в новый массив.
3. Обновить вместимость вектора.

```
void reserve(size_t new_capacity) {
    if (new_capacity > capacity) {
        auto new_data = new T[new_capacity];
        // TODO
    }
}
```

operator[], at()

operator[]:

1. Просто вернуть элемент вектора по заданному индексу.

at():

1. Проверить, что индекс находится в диапазоне от 0 до (длина вектора - 1).
2. Если индекс вне этого диапазона, бросить исключение std::out_of_range.
3. Вернуть элемент вектора по заданному индексу.

erase

erase():

1. Проверить, что индекс находится в диапазоне от 0 до (длина вектора - 1).
2. Если индекс вне этого диапазона, бросить исключение std::out_of_range.
3. Сдвинуть все элементы после указанного индекса на одну позицию влево.
4. Уменьшить длину вектора на 1.

assign(size_t count, const T& value):

1. Очистить содержимое вектора.
2. Если количество элементов count больше текущей вместимости вектора, увеличить вместимость.
3. Заполнить вектор count элементами, каждый из которых будет иметь значение value.
4. Установить длину вектора равной count.

assign(InputIt first, InputIt last):

1. Очистить содержимое вектора.
2. Посчитать количество элементов между first и last.
3. Если количество элементов больше текущей вместимости вектора, увеличить вместимость.
4. Проитерироваться от first до last и заполнить вектор значениями из заданного диапазона.
5. Установить длину вектора равной количеству элементов между first и last.

emplace_back(Args&&... args):

1. Проверить, не превышает ли текущая длина вектора его вместимость. Если превышает, увеличить вместимость вдвое.
2. Создать новый элемент в конце вектора, передав переданные аргументы (args) в конструктор этого элемента.
3. Увеличить длину вектора.

перегрузка оператора вывода вектора в консоль

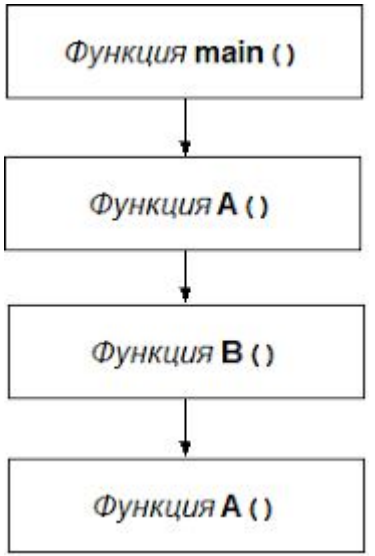
```
friend std::ostream& operator<<(std::ostream& os, const vector& vec) {
    os << "[";
    for (size_t i = 0; i < vec.length; ++i) {
        // TODO
    }
    os << "]";
    return os;
}
```

(DAM) Dynamic Attributes Methods

Функция main()

main() - главная функция в программе на C++. Она является точкой входа, где начинается выполнение программы, содержит основную логику программы и завершает ее, возвращая статус выполнения. Функция main() должна быть определена в вашей программе, иначе компилятор выдаст ошибку.

main() -> **foo0**() -> **foo1**() -> **foo2**() -> … ->**fooN**()



argc" и "argv" - это параметры командной строки, которые передаются в функцию "main" при запуске программы на языке C или C++.

- **"argc" (аргумент count)** - это целочисленная переменная, которая содержит количество аргументов командной строки, переданных при запуске программы, включая имя самой программы. Таким образом, **"argc" указывает на количество элементов в массиве "argv".**
- **"argv" (аргумент values)** - это массив строк (массив указателей на символы), который содержит сами аргументы командной строки. Первый элемент массива "argv[0]" обычно содержит имя исполняемого файла программы, а остальные элементы "argv[1]", "argv[2]", и т.д. содержат аргументы, переданные при запуске программы.

```
int main(int argc, char* argv[]) {..}
```

Пример вызова:

./app 1 2 3

```
iMac-andrey:cmake-build-release andrey$ ./untitled1
argc: 1
```

```
iMac-andrey:cmake-build-release andrey$ ./untitled1 1 2
argc: 3
argv[0]: ./untitled1
argv[1]: 1
```

pair

<pair>

- это структура данных, предоставляемая в стандартной библиотеке. Она позволяет объединить два объекта разных типов в один объект. Обычно используется, когда нужно вернуть два значения из функции или создать ассоциативный массив, например, когда необходимо связать ключ с значением.

IN STD:

```
template <typename T, typename U>
class pair {
   T first;
   U second;
   Pair() {}
   Pair(const T& f, const U& s)
   : first(f), second(s) {}
};
```

```
int main() {
   Std::pair<int, std::string> pair1(42, "Hello");
   Std::pair<int, int> pair2(42, 44);
   Std::pair<float, int> pair2(12.5, 100);
   // call first&second PAIR
   std::cout << "First element: " << pair1.first << std::endl;
   std::cout << "Second element: " << pair1.second <<
std::endl;
   return 0;
}
```

<pair call>

<pair call> понятие, которое означает вызов какой то функции или метода, который возвращает объект типа **pair** или его аналога, содержащего результаты выполнения функции.

```
Std::pair<int, int> divide(int dividend, int divisor) {
    return Std::pair<int, int>(dividend / divisor, dividend % divisor);
}
```

```
int main() {
    int dividend = 10, divisor = 3;
    auto pair = divide(dividend, divisor);
    std::cout << "Quotient: " << pair.first << std::endl;
    std::cout << "Remainder: " << pair.second << std::endl;
    return 0;
}
```

set (множество)

Что такое множество?

set - это структура данных, представляющая собой коллекцию уникальных элементов, в которой порядок элементов обычно не имеет значения.

Основные характеристики:

1. **Уникальность элементов:** Каждый элемент может присутствовать в множестве только один раз.
2. **Быстрый доступ и поиск:** Множества обеспечивают эффективный доступ и поиск элементов.
3. **Неупорядоченность:** Порядок элементов в множестве не гарантирован.
4. **Реализация различных операций:** Множества поддерживают операции добавления, удаления, поиска элементов, а также операции над множествами, такие как объединение, пересечение и разность.

базовый класс set

```
template<typename T>

class Set {

private:

   T* elements;

   int capacity, size;

public:

   Set(int initialCapacity = 10) : capacity(initialCapacity), size( 0) {elements = new
T[capacity];}

   ~Set() {delete[] elements;}
```

СОЗДАНИЕ МНОЖЕСТВА

```
Std::set<int> myset;
```

Методы работы со множеством

1. **insert:** Добавляет элемент в множество.
2. **erase:** Удаляет элемент из множества.
3. **find:** Ищет элемент в множестве.
4. **count:** Подсчитывает количество элементов с заданным значением.
5. **size:** Возвращает количество элементов в множестве.
6. **empty:** Проверяет, пусто ли множество.
7. **clear:** Очищает множество, удаляя все элементы.
8. **begin:** Возвращает итератор на начало множества.
9. **end:** Возвращает итератор на конец множества.
10. **lower_bound:** Возвращает итератор на первый элемент, не меньший, чем заданный.
11. **upper_bound:** Возвращает итератор на первый элемент, больший, чем заданный.
12. **equal_range:** Возвращает диапазон элементов с заданным значением.
13. **emplace:** Вставляет элемент в множество, используя конструктор для его построения на месте.
14. **emplace_hint:** Вставляет элемент в множество с использованием подсказки итератора.
15. **swap:** Обменивает содержимое двух множеств.
16. **key_comp:** Возвращает функцию сравнения, используемую для сравнения ключей.
17. **value_comp:** То же, что и key_comp, но принимает пару значений, а не ключей.
18. **get_allocator:** Возвращает аллокатор, используемый множеством.
19. **emplace_hint:** Вставляет элемент в множество с использованием подсказки итератора.
20. **operator=:** Перегруженный оператор присваивания, копирующий содержимое другого множества.
21. **operator==:** Перегруженный оператор равенства, сравнивающий два множества на равенство.
22. **operator!=:** Перегруженный оператор неравенства, сравнивающий два множества на неравенство.
23. **operator<:** Перегруженный оператор "меньше", сравнивающий множества лексикографически.
24. **operator<=:** Перегруженный оператор "меньше или равно", сравнивающий множества лексикографически.
25. **operator>:** Перегруженный оператор "больше", сравнивающий множества лексикографически.
26. **operator>=:** Перегруженный оператор "больше или равно", сравнивающий множества лексикографически.

insert

- Получить элемент, который нужно добавить в множество.
- Проверить, есть ли уже такой элемент в множестве.
- Если элемент уже есть, ничего не делать.
- Если элемента нет, добавить его в множество.

```
void insert(const T& element) {
        // Проверка наличия элемента в множестве
        for (int i = 0; i < size; ++i)
            if (elements[i] == element)
                return;
        // Расширение массива, если нужно
        if (size >= capacity) { resize();}
        // Добавление элемента
        elements[size++] = element;
    }
```

```
void resize() {
        // Увеличение размера массива в два раза
        capacity *= 2;
        T* newElements = new T[capacity];
        for (int i = 0; i < size; ++i)
            newElements[i] = elements[i];
        delete[] elements;
        elements = newElements;
    }
```

erase

- Получить элемент, который нужно удалить из множества.
- Найти этот элемент в множестве.
- Если элемент найден, удалить его из множества.

```
void erase(const T& element) {
    // Удаляем элемент из множества, если он присутствует
    for (int i = 0; i < size; ++i) {
        if (elements[i] == element) {
            // Сдвигаем все элементы после удаляемого влево
            for (int j = i; j < size - 1; ++j)
                elements[j] = elements[j + 1];
            --size;
            return;
        }
    }
}
```

find

- Получить элемент, который нужно найти в множестве.
- Начиная с начала множества, последовательно сравнивать каждый элемент с искомым.
- Если элемент найден, вернуть итератор на него.
- Если элемент не найден, вернуть итератор, указывающий на конец множества.

```
bool find(const T& element) const;
```

count/size

COUNT

- Получить элемент, количество вхождений которого нужно подсчитать в множестве.
- Начиная с начала множества, последовательно сравнивать каждый элемент с искомым.
- Подсчитывать количество совпадающих элементов.
- Вернуть количество совпадающих элементов.

SIZE

- Просто вернуть количество элементов в множестве.

```
int getSize() const { return size;}
```

empty/clear

empty:

- Проверить, пусто ли множество (т.е. не содержит ли оно ни одного элемента).

clear:

- Просто удалить все элементы из множества.

```
bool empty() const {
    return size == 0;
}

void clear() {
    // Очищаем множество, удаляя все элементы
    size = 0;
    std::cout << "Set has been cleared." << std::endl;
}
```

begin/end, lower_bound/upper_bound, equal_range, swap

begin:

- Вернуть итератор, указывающий на начало множества.

end:

- Вернуть итератор, указывающий на конец множества.

lower_bound:

- Вернуть итератор на первый элемент, который не меньше заданного.

upper_bound:

- Вернуть итератор на первый элемент, который больше заданного.

equal_range:

- Вернуть диапазон элементов с заданным значением.

swap:

- Обменять содержимое двух множеств.

```
void swap(Std::set<T>& other) {
    // Обмениваем содержимое двух множеств
    std::swap(elements, other.elements);
    std::swap(capacity, other.capacity);
    std::swap(size, other.size);
    std::cout << "Sets have been swapped." << std::endl;
}
```

operator=, operator==, operator !=

operator=:

- Присвоить одно множество другому, скопировав его содержимое.

operator==, operator!=

- Сравнить два множества на равенство, неравенство.

```
// Оператор присваивания
Std::set<T>& operator=(const Std::set<T>& other) {
   if (this != &other) {
       delete[] elements;
       capacity = other.capacity;
       size = other.size;
       elements = new T[capacity];
       for (int i = 0; i < size; ++i)
           elements[i] = other.elements[i];
   }
   return *this;
}
```

operator<, operator<=, operator>, operator>=

Выполнить операции сравнения <, <=, >, >=;

```
// Операторы сравнения
bool operator>(const Std::set<T>& other) const {
    return size > other.size;
}

bool operator>=(const Std::set<T>& other) const {
    return size >= other.size;
}
```

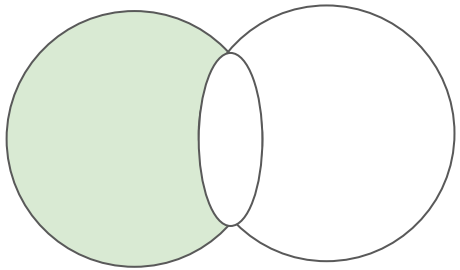
operator&; operator|

operator& (пересечение):

- Создать новое множество, которое будет содержать только элементы, которые присутствуют в обоих исходных множествах.
- Для каждого элемента в первом множестве проверить, присутствует ли он во втором множестве.
- Если элемент присутствует и в первом, и во втором множестве, добавить его в новое множество.

operator| (объединение):

- Создать новое множество, которое будет содержать все уникальные элементы из обоих исходных множеств.
- Добавить все элементы из первого множества в новое множество.
- Добавить все элементы из второго множества в новое множество, исключая дубликаты.



Operator&

```
// Оператор пересечения
Std::set<T> operator&(const Std::set<T>& other) const {
    Std::set<T> result;
    for (int i = 0; i < size; ++i) {
        if (other.find(elements[i])) {
            result.insert(elements[i]);
        }
    }
    return result;
}
```

Итераторы

Итераторы - это объекты, используемые для обхода элементов в контейнерах в языке программирования C++. Они предоставляют интерфейс для последовательного доступа к элементам коллекции и манипуляции ими.

Основные операции, которые можно выполнить с итераторами:

1. Перемещение: Итераторы могут перемещаться по элементам контейнера, позволяя получать доступ к каждому элементу в последовательном порядке.
2. Чтение и запись: Итераторы позволяют читать значения элементов контейнера и изменять их содержимое.
3. Указание на элемент: Итераторы предоставляют указатель на текущий элемент в контейнере.
4. Увеличение и уменьшение: Некоторые виды итераторов позволяют увеличивать и уменьшать свое значение, перемещаясь к следующему или предыдущему элементу в контейнере.

std::stack

https://en.cppreference.com/w/cpp/container/stack

std::stack - это контейнерный класс в стандартной библиотеке C++, который предоставляет интерфейс для работы с данными в структуре данных "стек". Стек работает по принципу "last-in, first-out" (LIFO), что означает, что последний элемент, помещенный в стек, будет первым, выталкиваемым из него.

1. **push(const T& val):** Добавляет элемент val на вершину стека.
2. **pop():** Удаляет элемент с вершины стека.
3. **top():** Возвращает ссылку на элемент на вершине стека. Стек при этом не изменяется.
4. **empty():** Проверяет, пуст ли стек. Возвращает true, если стек пустой, и false в противном случае.
5. **size():** Возвращает количество элементов в стеке.
6. **emplace(Args&&... args):** Создает и добавляет новый элемент на вершину стека, передавая аргументы args конструктору элемента.
7. **swap(stack& other):** Обменивает содержимое двух стеков. Текущий стек становится таким же, как other, а other становится пустым.
8. **operator=:** Присваивает содержимое одного стека другому.
9. **operator==:** Сравнивает содержимое двух стеков на равенство.
10. **operator!=:** Сравнивает содержимое двух стеков на неравенство.
11. **operator<:** Сравнивает содержимое двух стеков лексикографически (сравнивает каждый элемент от вершины к основанию).
12. **operator<=:** Сравнивает содержимое двух стеков на меньше или равно.
13. **operator>:** Сравнивает содержимое двух стеков на больше.
14. **operator>=:** Сравнивает содержимое двух стеков на больше или равно.
15. **std::swap(stack& x, stack& y):** Обменивает содержимое двух стеков x и y

Пример кода

```
#include <iostream>
#include <stack>

int main() {
   std::stack<int> s;
   // Добавление элементов в стек
   s.push(5);
   s.push(10);
   s.push(15);
   // Вывод размера стека
   std::cout << "Размер стека: " << s.size() << std::endl;
   // Вывод и удаление вершины стека
   std::cout << "Вершина стека: " << s.top() << std::endl;
   s.pop();
   // Проверка пустоты стека
   if (!s.empty()) {
       std::cout << "Стек не пустой." << std::endl;
   } else {std::cout << "Стек пустой." << std::endl;}
   return 0;
}
```

std::queue

https://en.cppreference.com/w/cpp/container/queue

std::queue - контейнерный класс в стандартной библиотеке C++, который предоставляет интерфейс для работы с данными в структуре данных "очередь". Очередь работает по принципу "first-in, first-out" (FIFO), что означает, что первый элемент, помещенный в очередь, будет первым, выталкиваемым из нее.

1. **push(const T& value):** Добавляет элемент в конец очереди.
2. **emplace(Args&&... args):** Создает и добавляет элемент в конец очереди, используя переданные аргументы для конструирования объекта.
3. **pop():** Удаляет первый элемент из начала очереди.
4. **front():** Возвращает ссылку на первый элемент очереди.
5. **back():** Возвращает ссылку на последний элемент очереди.
6. **empty():** Проверяет, пуста ли очередь.
7. **size():** Возвращает количество элементов в очереди.
8. **swap(queue& other):** Обменивает содержимое текущей очереди с содержимым переданной очереди other.
9. **operator=:** Присваивает содержимое одной очереди другой.
10. **emplace_back():** Вставляет новый элемент в конец очереди.
11. **emplace_front():** Вставляет новый элемент в начало очереди.
12. **clear():** Удаляет все элементы из очереди.
13. **at(size_type n):** Возвращает ссылку на элемент с указанным индексом в очереди.
14. **operator[]:** Позволяет обратиться к элементу очереди по индексу.
15. **get_allocator():** Возвращает аллокатор, используемый для управления памятью элементов в очереди.

Пример кода

```
#include <iostream>
#include <queue>

int main() {
   std::queue<int> q;
   // Добавление элементов в очередь
   q.push(5);
   q.push(10);
   q.push(15);
   // Вывод размера очереди
   std::cout << "Размер очереди: " << q.size() << std::endl;
   // Вывод и удаление первого элемента очереди
   std::cout << "Первый элемент очереди: " << q.front() << std::endl;
   q.pop();
   // Проверка пустоты очереди
   if (!q.empty()) {
       std::cout << "Очередь не пустая." << std::endl;
   } else { std::cout << "Очередь пустая." << std::endl;}
   return 0;
}
```

std::deque

https://en.cppreference.com/w/cpp/container/deque

std::deque (double-ended queue) - это контейнер в стандартной библиотеке C++, который представляет собой двустороннюю очередь, то есть контейнер, который поддерживает вставку и удаление элементов как с начала, так и с конца.

1. push_back(const T& value): Добавляет элемент value в конец очереди.
2. push_front(const T& value): Добавляет элемент value в начало очереди.
3. pop_back(): Удаляет элемент с конца очереди.
4. pop_front(): Удаляет элемент с начала очереди.
5. emplace_back(Args&&... args): Создает и добавляет элемент в конец очереди с использованием конструктора элемента и переданных аргументов.
6. emplace_front(Args&&... args): Создает и добавляет элемент в начало очереди с использованием конструктора элемента и переданных аргументов.
7. back(): Возвращает ссылку на последний элемент в очереди.
8. front(): Возвращает ссылку на первый элемент в очереди.
9. at(size_type pos): Возвращает ссылку на элемент в указанной позиции в очереди. Генерирует исключение std::out_of_range, если pos выходит за пределы допустимых индексов.
10. size() const noexcept: Возвращает количество элементов в очереди.
11. empty() const noexcept: Проверяет, пуста ли очередь. Возвращает true, если очередь пуста, и false в противном случае.
12. clear(): Удаляет все элементы из очереди.
13. resize(size_type count, const T& value = T()): Изменяет размер очереди, заполняя новые элементы значением value, если count больше текущего размера. Если count меньше текущего размера, элементы с конца очереди удаляются.
14. erase(iterator position): Удаляет элемент в указанной позиции из очереди.
15. insert(iterator position, const T& value): Вставляет элемент value в указанную позицию в очереди.

Пример кода

```cpp
#include <iostream>
#include <deque>
#include <string>

int main() {
   std::deque<std::string> d;
   // Добавление элементов в начало и конец очереди
   d.push_back("World");
   d.push_front("Hello");
   // Вывод элементов очереди
   std::cout << "Первый элемент: " << d.front() << std::endl;
   std::cout << "Последний элемент: " << d.back() << std::endl;
   // Удаление элементов из начала и конца очереди
   d.pop_front();
   d.pop_back();
   // Проверка пустоты очереди
   if (!d.empty()) {
       std::cout << "Очередь не пустая." << std::endl;
   } else {std::cout << "Очередь пустая." << std::endl;}
   return 0;
}
```

std::list

https://en.cppreference.com/w/cpp/container/list

std::list - это контейнер из стандартной библиотеки C++, который представляет собой двусвязный список элементов. Этот контейнер обеспечивает эффективную вставку и удаление элементов в любом месте списка, за счет того, что каждый элемент хранит указатели на предыдущий и следующий элементы.

Методы работы с std::list:

1. std::list::list() - конструктор по умолчанию, создает пустой список.
2. std::list::list(size_type count, const T& value) - конструктор создает список, содержащий count элементов, каждый из которых равен value.
3. std::list::begin() - возвращает итератор на начало списка.
4. std::list::end() - возвращает итератор на конец списка.
5. std::list::push_back(const T& value) - добавляет элемент value в конец списка.
6. std::list::push_front(const T& value) - добавляет элемент value в начало списка.
7. std::list::pop_back() - удаляет последний элемент из списка.
8. std::list::pop_front() - удаляет первый элемент из списка.
9. std::list::insert(iterator pos, const T& value) - вставляет элемент value перед элементом, на который указывает итератор pos.
10. std::list::erase(iterator pos) - удаляет элемент, на который указывает итератор pos.
11. std::list::size() - возвращает количество элементов в списке.
12. std::list::empty() - проверяет, пуст ли список.
13. std::list::clear() - удаляет все элементы из списка, делая его пустым.
14. std::list::resize(size_type count, const T& value) - изменяет размер списка на count, добавляя или удаляя элементы. Если count больше текущего размера списка, новые элементы будут добавлены и заполнены значением value, если он указан.
15. std::list::reverse() - изменяет порядок элементов в списке на противоположный.

пример кода

```cpp
#include <iostream>
#include <list>

int main() {
    // Создаем пустой список
    std::list<int> myList;

    // Добавляем элементы в список
    myList.push_back(1);
    myList.push_back(2);
    myList.push_back(3);
    myList.push_back(4);
    myList.push_back(5);

    // Выводим элементы списка на экран
    std::cout << "std::list:\n";
    for (const auto& num : myList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

std::forward_list

https://en.cppreference.com/w/cpp/container/forward_list

std::forward_list - это контейнер из стандартной библиотеки C++, который представляет собой односвязный список элементов. В отличие от std::list, в котором каждый элемент хранит указатели на предыдущий и следующий элементы, в std::forward_list каждый элемент хранит только указатель на следующий элемент.

Методы работы с std::forward_list:

1. std::forward_list::forward_list() - конструктор по умолчанию, создает пустой список.
2. std::forward_list::forward_list(size_type count, const T& value) - конструктор создает список, содержащий count элементов, каждый из которых равен value.
3. std::forward_list::begin() - возвращает итератор на начало списка.
4. std::forward_list::end() - возвращает итератор на конец списка.
5. std::forward_list::push_front(const T& value) - добавляет элемент value в начало списка.
6. std::forward_list::pop_front() - удаляет первый элемент из списка.
7. std::forward_list::insert_after(iterator pos, const T& value) - вставляет элемент value после элемента, на который указывает итератор pos.
8. std::forward_list::erase_after(iterator pos) - удаляет элемент, следующий за элементом, на который указывает итератор pos.
9. std::forward_list::size() - возвращает количество элементов в списке.
10. std::forward_list::empty() - проверяет, пуст ли список.
11. std::forward_list::clear() - удаляет все элементы из списка, делая его пустым.
12. std::forward_list::resize(size_type count, const T& value) - изменяет размер списка на count, добавляя или удаляя элементы. Если count больше текущего размера списка, новые элементы будут добавлены и заполнены значением value, если он указан.
13. std::forward_list::reverse() - изменяет порядок элементов в списке на противоположный.
14. std::forward_list::merge(std::forward_list& other) - объединяет два отсортированных списка в один отсортированный список. Оба списка должны быть отсортированы перед объединением.
15. std::forward_list::sort() - сортирует элементы списка в порядке возрастания.

пример кода

```cpp
#include <iostream>
#include <forward_list>

int main() {
    // Создаем пустой односвязный список
    std::forward_list<int> myForwardList;

    // Добавляем элементы в список
    myForwardList.push_front(1);
    myForwardList.push_front(2);
    myForwardList.push_front(3);
    myForwardList.push_front(4);
    myForwardList.push_front(5);

    // Выводим элементы списка на экран
    std::cout << "std::forward_list:\n";
    for (const auto& num : myForwardList) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

std::vector

https://en.cppreference.com/w/cpp/container/vector

std::vector - это контейнер из стандартной библиотеки C++, который представляет собой динамический массив элементов. Он обеспечивает быстрый доступ к элементам по индексу и динамическое изменение размера массива.

Методы работы с std::vector:

1. std::vector::vector() - конструктор по умолчанию, создает пустой вектор.
2. std::vector::vector(size_type count, const T& value) - конструктор создает вектор, содержащий count элементов, каждый из которых равен value.
3. std::vector::begin() - возвращает итератор на начало вектора.
4. std::vector::end() - возвращает итератор на конец вектора.
5. std::vector::push_back(const T& value) - добавляет элемент value в конец вектора.
6. std::vector::pop_back() - удаляет последний элемент из вектора.
7. std::vector::insert(iterator pos, const T& value) - вставляет элемент value перед элементом, на который указывает итератор pos.
8. std::vector::erase(iterator pos) - удаляет элемент, на который указывает итератор pos.
9. std::vector::size() - возвращает количество элементов в векторе.
10. std::vector::empty() - проверяет, пуст ли вектор.
11. std::vector::clear() - удаляет все элементы из вектора, делая его пустым.
12. std::vector::resize(size_type count, const T& value) - изменяет размер вектора на count, добавляя или удаляя элементы. Если count больше текущего размера вектора, новые элементы будут добавлены и заполнены значением value, если он указан.
13. std::vector::reserve(size_type count) - увеличивает емкость вектора до count элементов, чтобы уменьшить количество перераспределений памяти.
14. std::vector::shrink_to_fit() - уменьшает емкость вектора так, чтобы она соответствовала количеству элементов в векторе.
15. std::vector::swap(std::vector& other) - обменивает содержимое двух векторов.

пример кода

```cpp
#include <iostream>
#include <vector>

int main() {
    // Создаем пустой вектор
    std::vector<int> myVector;

    // Добавляем элементы в вектор
    myVector.push_back(1);
    myVector.push_back(2);
    myVector.push_back(3);
    myVector.push_back(4);
    myVector.push_back(5);

    // Выводим элементы вектора на экран
    std::cout << "std::vector:\n";
    for (const auto& num : myVector) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

std::set/map

https://en.cppreference.com/w/cpp/container/set
https://en.cppreference.com/w/cpp/container/map

std::unordered_set и std::unordered_map - это контейнеры из стандартной библиотеки C++, которые предоставляют альтернативу std::set и std::map соответственно. Они основаны на хэшировании, что обеспечивает более эффективный доступ к элементам, чем в случае деревьев поиска.

1. **unordered_set:**
   - std::unordered_set представляет собой множество уникальных элементов, но в отличие от std::set, элементы не автоматически сортируются.
   - Вставка, удаление и поиск элементов в std::unordered_set выполняются в среднем за время O(1) в лучшем случае, худший случай может быть O(n).
   - Этот контейнер особенно полезен, когда требуется быстрый доступ к элементам без необходимости их сортировки или когда порядок элементов не имеет значения.
2. **unordered_map:**
   - std::unordered_map представляет собой ассоциативный массив, где каждый ключ связан с уникальным значением, но в отличие от std::map, элементы не автоматически сортируются по ключу.
   - Вставка, удаление и поиск элементов в std::unordered_map также выполняются в среднем за время O(1) в лучшем случае, худший случай может быть O(n).
   - Этот контейнер часто используется, когда требуется быстрый доступ к данным по ключу без необходимости их сортировки или когда порядок ключей не важен.

Для std::set:

1. std::set::set() - конструктор по умолчанию, создает пустой set.
2. std::set::set(const std::initializer_list<T>& init) - конструктор, создающий set из списка инициализации.
3. std::set::begin() - возвращает итератор на начало множества.
4. std::set::end() - возвращает итератор на конец множества.
5. std::set::insert(const T& value) - добавляет элемент value в множество.
6. std::set::erase(const T& value) - удаляет элемент value из множества.
7. std::set::find(const T& value) - ищет элемент value в множестве. Если элемент найден, возвращает итератор на него, иначе возвращает итератор на конец множества.
8. std::set::size() - возвращает количество элементов в множестве.
9. std::set::empty() - проверяет, пусто ли множество.
10. std::set::clear() - удаляет все элементы из множества, делая его пустым.
11. std::set::lower_bound(const T& value) - возвращает итератор на первый элемент в множестве, который не меньше value.
12. std::set::upper_bound(const T& value) - возвращает итератор на первый элемент в множестве, который больше value.
13. std::set::count(const T& value) - возвращает количество элементов с определенным значением в множестве (обычно 0 или 1 для set).
14. std::set::emplace(Args&&... args) - создает новый элемент в множестве и вставляет его, не создавая копию, если он не существует.
15. std::set::emplace_hint(const_iterator hint, Args&&... args) - аналогично emplace, но использует подсказку hint для оптимизации вставки.

пример кода

```
#include <iostream>
#include <set>

int main() {
    // Создаем пустое множество
    std::set<int> mySet;

    // Добавляем элементы в множество
    mySet.insert(5);
    mySet.insert(2);
    mySet.insert(7);
    mySet.insert(1);
    mySet.insert(3);

    // Выводим элементы множества на экран
    std::cout << "std::set:\n";
    for (const auto& num : mySet) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Для std::map:

1. std::map::map() - конструктор по умолчанию, создает пустой map.
2. std::map::map(const std::initializer_list<std::pair<const Key, T>>& init) - конструктор, создающий map из списка инициализации пар ключ-значение.
3. std::map::begin() - возвращает итератор на начало словаря.
4. std::map::end() - возвращает итератор на конец словаря.
5. std::map::insert(const std::pair<const Key, T>& pair) - добавляет пару ключ-значение в словарь.
6. std::map::erase(const Key& key) - удаляет элемент с заданным ключом из словаря.
7. std::map::find(const Key& key) - ищет элемент с заданным ключом в словаре. Если элемент найден, возвращает итератор на него, иначе возвращает итератор на конец словаря.
8. std::map::size() - возвращает количество элементов в словаре.
9. std::map::empty() - проверяет, пуст ли словарь.
10. std::map::clear() - удаляет все элементы из словаря, делая его пустым.
11. std::map::lower_bound(const Key& key) - возвращает итератор на первый элемент в словаре, который не меньше key.
12. std::map::upper_bound(const Key& key) - возвращает итератор на первый элемент в словаре, который больше key.
13. std::map::count(const Key& key) - возвращает количество элементов с определенным ключом в словаре (обычно 0 или 1 для map).
14. std::map::emplace(Args&&... args) - создает новую пару ключ-значение в словаре и вставляет ее, не создавая копию, если она не существует.
15. std::map::emplace_hint(const_iterator hint, Args&&... args) - аналогично emplace, но использует подсказку hint для оптимизации вставки.

пример кода

```cpp
#include <iostream>
#include <map>

int main() {
    // Создаем пустой словарь
    std::map<int, std::string> myMap;

    // Добавляем пары ключ-значение в словарь
    myMap[1] = "one";
    myMap[2] = "two";
    myMap[3] = "three";
    myMap[4] = "four";
    myMap[5] = "five";

    // Выводим элементы словаря на экран
    std::cout << "std::map:\n";
    for (const auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second << "\n";
    }

    return 0;
}
```

std::unordered_set/map

https://en.cppreference.com/w/cpp/container/unordered_set
https://en.cppreference.com/w/cpp/container/unordered_map

Для std::unordered_map:

- unordered_map::begin(): Возвращает итератор на начало контейнера.
- unordered_map::end(): Возвращает итератор на конец контейнера.
- unordered_map::cbegin(): Возвращает константный итератор на начало контейнера.
- unordered_map::cend(): Возвращает константный итератор на конец контейнера.
- unordered_map::empty(): Проверяет, пуст ли контейнер.
- unordered_map::size(): Возвращает количество элементов в контейнере.
- unordered_map::max_size(): Возвращает максимальное количество элементов, которое контейнер может содержать.
- unordered_map::clear(): Очищает контейнер, удаляя все элементы.
- unordered_map::find(const key_type& k): Поиск элемента с заданным ключом k.
- unordered_map::count(const key_type& k): Возвращает количество элементов с заданным ключом k.
- unordered_map::erase(const key_type& k): Удаляет элемент с ключом k.
- unordered_map::insert(const value_type& v): Вставляет элемент v в контейнер.
- unordered_map::emplace(Args&&... args): Создает и вставляет элемент, используя переданные аргументы.
- unordered_map::bucket_count(): Возвращает количество "ведер" (бакетов) в контейнере.
- unordered_map::load_factor(): Возвращает текущий коэффициент заполнения контейнера.

Для std::unordered_set:

- unordered_set::begin(): Возвращает итератор на начало контейнера.
- unordered_set::end(): Возвращает итератор на конец контейнера.
- unordered_set::cbegin(): Возвращает константный итератор на начало контейнера.
- unordered_set::cend(): Возвращает константный итератор на конец контейнера.
- unordered_set::empty(): Проверяет, пуст ли контейнер.
- unordered_set::size(): Возвращает количество элементов в контейнере.
- unordered_set::max_size(): Возвращает максимальное количество элементов, которое контейнер может содержать.
- unordered_set::clear(): Очищает контейнер, удаляя все элементы.
- unordered_set::find(const key_type& k): Поиск элемента с заданным ключом k.
- unordered_set::count(const key_type& k): Возвращает количество элементов с заданным ключом k.
- unordered_set::erase(const key_type& k): Удаляет элемент с ключом k.
- unordered_set::insert(const value_type& v): Вставляет элемент v в контейнер.
- unordered_set::emplace(Args&&... args): Создает и вставляет элемент, используя переданные аргументы.
- unordered_set::bucket_count(): Возвращает количество "ведер" (бакетов) в контейнере.
- unordered_set::load_factor(): Возвращает текущий коэффициент заполнения контейнера.

Пример кода unordered_set (вставка)

```
// Пример использования std::unordered_set
std::unordered_set<int> mySet;

// Вставка элементов
mySet.insert(5);
mySet.insert(3);
mySet.insert(8);

// Проверка наличия элемента
if (mySet.find(3) != mySet.end()) {
    std::cout << "Элемент 3 найден в множестве. \n";
}
```

Пример кода unordered_map (вставка)

```
// Пример использования std::unordered_map
std::unordered_map<std::string, int> myMap;

// Вставка пар ключ-значение
myMap["apple"] = 5;
myMap["banana"] = 3;
myMap["orange"] = 8;

// Получение значения по ключу
std::cout << "Значение для ключа 'banana': " << myMap["banana"] << "\n";
```

Пример кода unordered_set (удаление)

```
// Пример использования std::unordered_set
std::unordered_set<int> mySet = {5, 3, 8, 10};

// Удаление элемента из множества
mySet.erase(3);

// Проверка наличия элемента после удаления
if (mySet.find(3) == mySet.end()) {
    std::cout << "Элемент 3 отсутствует в множестве.\n";
}
```

Пример кода unordered_map (удаления)

```
// Пример использования std::unordered_map
std::unordered_map<std::string, int> myMap = {{"apple", 5}, {"banana", 3},
{"orange", 8}};

// Удаление элемента из словаря
myMap.erase("banana");

// Проверка наличия ключа после удаления
if (myMap.find("banana") == myMap.end()) {
    std::cout << "Ключ 'banana' отсутствует в словаре.\n";
}
```

Unordered_set / Unordered_map

ВАЖНО!!!

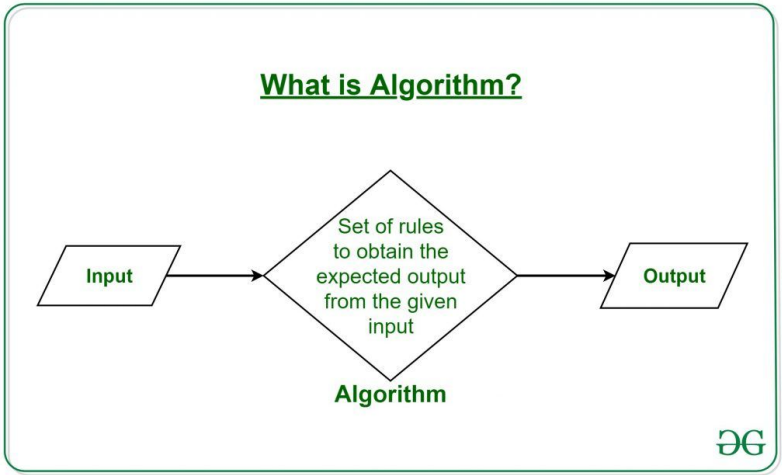
КАСАЕМО СОРТИРОВКИ ДАННЫХ;

std::unordered_set и std::unordered_map не поддерживают операцию сортировки, так как их основное предназначение - быстрый доступ к данным с использованием хэширования, а не сохранение порядка элементов.

```
#include <algorithm>
```

ALGORITHM…

#include <algorithm> - это заголовочный файл из стандартной библиотеки C++, который содержит реализации различных алгоритмов для работы с контейнерами, итераторами и другими структурами данных. Этот заголовочный файл предоставляет широкий спектр стандартных алгоритмов, таких как сортировка, поиск, перестановка элементов и многое другое.



Методы работы;

- **std::sort** - сортирует элементы в указанном диапазоне.
- **std::find** - находит первый элемент в диапазоне, равный заданному значению.
- **std::reverse** - обращает порядок элементов в указанном диапазоне.
- **std::accumulate** - суммирует элементы в указанном диапазоне.
- **std::copy** - копирует элементы из одного диапазона в другой.
- **std::for_each** - применяет функцию к каждому элементу в указанном диапазоне.
- **std::unique** - удаляет все последовательные дублирующиеся элементы в диапазоне.
- **std::count** - подсчитывает количество элементов в диапазоне, равных заданному значению.
- **std::binary_search** - проверяет, содержит ли отсортированный диапазон элемент с заданным значением.
- **std::lower_bound** - возвращает итератор на первый элемент, не меньший заданного значения.
- **std::upper_bound** - возвращает итератор на первый элемент, больший заданного значения.
- **std::equal** - проверяет, равны ли два диапазона.
- **std::transform** - применяет функцию к элементам в диапазоне и сохраняет результат в другой диапазон.
- **std::partition** - разделяет диапазон на две части: удовлетворяющие и не удовлетворяющие предикату.
- **std::merge** - сливает два отсортированных диапазона в один.
- **std::min_element** - находит минимальный элемент в диапазоне.
- **std::max_element** - находит максимальный элемент в диапазоне.
- **std::fill** - заполняет все элементы в диапазоне заданным значением.
- **std::fill_n** - заполняет заданное количество элементов значением.
- **std::remove** - удаляет все элементы, равные заданному значению, и возвращает новую границу диапазона.
- **std::remove_if** - удаляет все элементы, удовлетворяющие предикату.
- **std::replace** - заменяет все элементы, равные заданному значению, новым значением.
- **std::replace_if** - заменяет все элементы, удовлетворяющие предикату, новым значением.
- **std::rotate** - перемещает элементы в диапазоне так, что элемент, указываемый первым, становится началом нового диапазона.
- **std::nth_element** - частично сортирует диапазон так, что элемент на n-й позиции будет тем, каким он был бы в отсортированном массиве.
- **std::shuffle** - случайным образом переставляет элементы в диапазоне.
- **std::clamp** - ограничивает значение диапазоном [low, high].
- **std::includes** - проверяет, является ли один отсортированный диапазон подмножеством другого.
- **std::set_union** - вычисляет объединение двух отсортированных диапазонов.
- **std::set_intersection** - вычисляет пересечение двух отсортированных диапазонов.
- **std::set_difference** - вычисляет разность двух отсортированных диапазонов.

STL - др. компоненты

std::chrono

std::chrono – это библиотека для работы с временем и таймерами. Она предоставляет удобные средства для работы с временными интервалами и точками времени. Основные компоненты std::chrono включают:

Durations (продолжительности):

- Представляют собой временные интервалы.
- Могут быть определены с различной точностью, например, секунды (std::chrono::seconds), миллисекунды (std::chrono::milliseconds), микросекунды (std::chrono::microseconds) и т.д.

```
std::chrono::seconds sec(1); // 1 секунда
std::chrono::milliseconds ms(100); // 100 миллисекунд
```

std::chrono

Time points (точки времени):

- Представляют собой конкретные моменты времени.
- Обычно используются вместе с часовыми типами, такими как std::chrono::system_clock или std::chrono::steady_clock.

```
auto now = std::chrono::system_clock::now(); // текущий момент времени
```

Clocks (часы):

- Представляют собой источники времени, такие как системные часы или монотонные часы.

```
auto start = std::chrono::steady_clock::now(); // начало отсчета времени
```

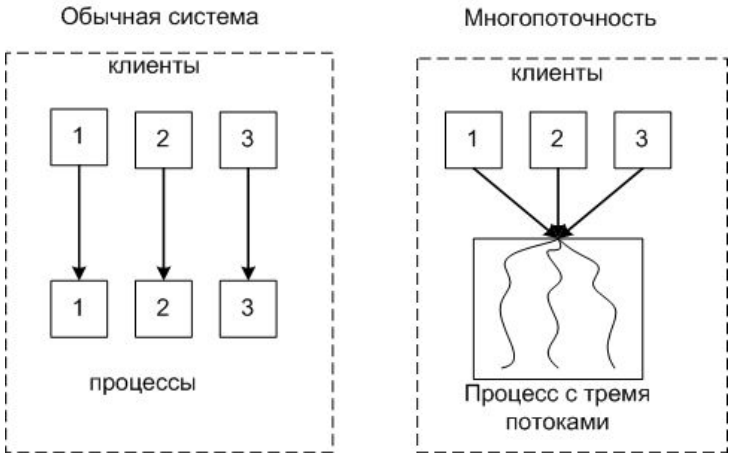
std::chrono - полный пример;

```
#include <iostream>
#include <chrono>
#include <thread>

void foo() {
    using namespace std::chrono;
    auto start = steady_clock::now();
    std::this_thread::sleep_for(seconds(1)); // пауза на 1 секунду
    auto end = steady_clock::now();
    duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << " seconds\n";
}
```

потоки.

Поток (англ. "thread") — это единица выполнения внутри процесса. В контексте программирования потоки позволяют выполнять несколько задач параллельно внутри одного процесса. Каждый поток имеет свой собственный стек, регистры и локальные переменные, но разделяет память и ресурсы процесса с другими потоками.



Зачем нужны потоки?

1. **Параллельное выполнение задач**: Потоки позволяют выполнять несколько задач одновременно, что может существенно ускорить выполнение программ, особенно на многоядерных процессорах.
2. **Реактивность и отзывчивость**: В пользовательских приложениях потоки позволяют выполнять фоновые задачи без блокировки основного пользовательского интерфейса, что делает приложение более отзывчивым.
3. **Увеличение производительности**: В некоторых задачах, таких как обработка больших объемов данных или выполнение сложных вычислений, потоки могут использоваться для распределения работы между несколькими ядрами процессора, что может повысить производительность.
4. **Сокрытие латентности ввода-вывода**: Потоки можно использовать для выполнения задач ввода-вывода (например, чтения/записи файлов, сетевого ввода-вывода) в фоновом режиме, тем самым позволяя основной программе продолжать работу.

многопоточность;

Многопоточность – это способность программы или операционной системы выполнять несколько потоков (нитей) параллельно. В контексте современных вычислительных систем многопоточность используется для повышения производительности и эффективности программ за счет параллельного выполнения задач. Рассмотрим подробнее концепции, преимущества и сложности многопоточности.

Основные концепции многопоточности

1. **Поток (thread)**:
   - Поток – это наименьшая единица обработки, которую планировщик операционной системы может управлять.
   - Потоки внутри одного процесса разделяют общую память и ресурсы, но имеют свои собственные регистры и стек.
2. **Процесс (process)**:
   - Процесс – это программа в состоянии выполнения.
   - Каждый процесс имеет свое собственное адресное пространство и ресурсы.
   - Процессы могут содержать несколько потоков.
3. **Многопоточность (multithreading)**:
   - Многопоточность позволяет одному процессу выполнять несколько потоков параллельно.

плюсы многопоточности ++

Повышение производительности:

- За счет распараллеливания задач, многопоточность позволяет более эффективно использовать ресурсы многоядерных процессоров.

Повышение отзывчивости:

- В графических и пользовательских интерфейсах многопоточность позволяет выполнять длительные операции в фоновом режиме, сохраняя при этом отзывчивость интерфейса.

Улучшение ресурсоемкости:

- Многопоточность позволяет выполнять ввод-вывод и другие блокирующие операции в отдельных потоках, что улучшает общую ресурсоемкость приложения.

проблемы многопоточности;

Синхронизация:

- Потоки, разделяющие общие ресурсы, должны быть синхронизированы, чтобы избежать гонок данных и состояний гонок.
- Для этого используются мьютексы (std::mutex), блокировки (std::lock_guard), условные переменные (std::condition_variable) и атомарные операции (std::atomic).

Дедлоки (взаимные блокировки):

- Дедлок возникает, когда два или более потоков зацикливаются в ожидании ресурсов, удерживаемых друг другом.
- Это приводит к тому, что потоки никогда не завершат выполнение.

Сложность отладки и тестирования:

- Многопоточные программы сложнее отлаживать и тестировать из-за непредсказуемого характера переключения потоков и взаимодействия между ними.

Производительность:

- Переключение контекста между потоками и использование механизмов синхронизации может ввести накладные расходы, что может негативно сказаться на производительности, если не использовать многопоточность правильно.

виды потоков.

Потоки на уровне ядра (Kernel-level threads):

- Управляются операционной системой.
- Каждый поток отображается на реальный поток операционной системы.
- Примеры: POSIX threads (pthreads) на Unix-подобных системах, Windows threads.

Пользовательские потоки (User-level threads):

- Управляются библиотеками или самим приложением, без вмешательства операционной системы.
- Легче создавать и управлять, так как переключение контекста происходит в пользовательском пространстве.
- Однако они не могут использовать преимущества многоядерных процессоров без явной поддержки от операционной системы.
- Пример: Зелёные потоки (Green threads).

Гибридные потоки (Hybrid threads):

- Комбинируют пользовательские и ядровые потоки.
- Пример: Потоки в Java Virtual Machine (JVM), которые могут быть реализованы как гибридные потоки в зависимости от реализации JVM и платформы.

std::thread

std::thread – это класс для создания и управления потоками выполнения. С помощью std::thread можно создавать параллельные задачи и синхронизировать их выполнение. Основные функции включают:

Создание потока:

- Поток создается с помощью конструктора, которому передается вызываемая сущность (например, функция или объект с перегруженным оператором ()).

```
void threadFunction() {
    std::cout << "Thread is running\n";
}

int main() {
    std::thread t(threadFunction); // создаем поток
    t.join(); // ждем завершения потока
    return 0;
}
```

std::thread

Синхронизация:

- join(): ожидание завершения потока.
- detach(): отделение потока для самостоятельного выполнения.

```
void detachedFunction() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Detached thread finished\n";
}

int main() {
    std::thread t(detachedFunction);
    t.detach(); // отделяем поток
    std::this_thread::sleep_for(std::chrono::seconds(2)); // ждем, чтобы поток завершился
    return 0;
}
```

std::atomic

std::atomic – это шаблонный класс, обеспечивающий атомарные операции с переменными. Это значит, что все операции с такими переменными выполняются как единое целое, без возможности прерывания другими потоками. Основные функции включают:

1. **Создание атомарных переменных**:
    - Поддерживаются различные типы данных, такие как int, bool, указатели и др.
    - ***Пример***: `std::atomic<int> counter(0);`

std::atomic

Атомарные операции:

- Операции чтения, записи, увеличения, уменьшения и другие выполняются атомарно.

```
std::atomic<int> counter(0);

void increment() {
   for (int i = 0; i < 1000; ++i) ++counter;
}

int main() {
   std::thread t1(increment);
   std::thread t2(increment);
   t1.join();
   t2.join();
   std::cout << "Final counter value: " << counter << '\n';
   return 0;
}
```