Язык программирования C++

Алгоритмы и предикаты.

Преподаватели:
Пысин Максим Дмитриевич, ассистент кафедры ИКТ
Краснов Дмитрий Олегович, аспирант кафедры ИКТ
Лобанов Алексей Владимирович, аспирант кафедры ИКТ
Крашенинников Роман Сергеевич, аспирант кафедры ИКТ


Шо, опять?!

Стандартная библиотека

В языке программирования C++ термин **Стандартная Библиотека** означает коллекцию классов и функций, написанных на самом языке.
Стандартные C++ библиотеки представляют собой наборы функций, констант, классов, объектов и шаблонов, которые расширяют язык C++ предоставляя базовую функциональность для выполнения различных задач, таких как: классы для взаимодействия с операционной системой, контейнеры данных, манипуляторы для работы с этими данными и наиболее используемые алгоритмы.
Все элементы стандартных библиотек C++ распределены по различным заголовочным файлам, которые необходимо подключать к программе. Только так можно использовать элементы этих библиотек. Ниже приведены стандартные библиотеки языка программирования C++.
Стандартная Библиотека поддерживает несколько основных контейнеров, функций для работы с этими контейнерами, объектов-функции, основных типов строк и потоков (включая интерактивный и файловый ввод-вывод), поддержку некоторых языковых особенностей, и часто используемые функции для выполнения таких задач, как, например, нахождение квадратного корня числа.
Стандартная библиотека шаблонов (STL) — подмножество стандартной библиотеки C++ и содержит контейнеры, алгоритмы, итераторы, объекты-функции и т. д. Хотя некоторые программисты используют термин «STL» вместе (или попеременно) с термином «Стандартная библиотека C++».

Состав стандартной библиотеки

В стандартной бибилиотеке C++ выделяют следующие компоненты:

- **Language support library** – отвечает за языковую поддержку
- **Diagnostics library** – отвечает за исключения(ошибки времени исполнения)
- **General utilities library** – отвечает за утилиты общего характера
- **Strings library** – отвечает за работу со строками
- **Localization library** – отвечает за локализацию программы
- **Containers library** – отвечает за все варианты контейнеров и адаптеров
- **Iterators library** – отвечает за итераторы для работы с контейнерами
- **Algorithms library** – отвечает за стандартные алгоритмы
- **Numerics library** – отвечает за работу с числами
- **Input/output library** – отвечает за ввод/вывод информации
- **Regular expressions library** – отвечает за регулярные выражения
- **Atomic operations library** – отвечает за атомарные операции
- **Thread support library** – отвечает за многопоточность

В стандартной библиотеке шаблонов выделяют пять основных компонентов:

- **Контейнер** (container)
- **Итератор** (iterator)
- **Алгоритмы** (algorithm)
- **Адаптеры** (adaptor)
- **Функциональные объекты** (functor)

Библиотека алгоритмов

Алгоритм — конечная совокупность точно заданных правил решения произвольного класса задач или набор инструкций, описывающих порядок действий исполнителя для решения некоторой задачи.

Методы сортировки коллекции:

- sort - сортировка,
- stable_sort – стабильная сортировка,
- partial_sort – частичная сортировка,
- partial_sort_copy – частичная сортировка с копированием,
- nth_element – групировка элементов больше и меньше определенного

Работа с отсортированными коллекциями:

- binary_search – бинарный поиск
- lower_bound – поиск первого не меньшего элемента упорядоченной последовательности
- upper_bound – поиск первого большего элемента упорядоченной последовательности
- merge – объединение отсортированных последовательностей и запись в место указанное для результата
- inplace_merge – объединение последовательностей и запись в начало объеденяемых последовательностей
- includes – проверяет включает ли одна последовательность другую
- min – минимальное значение
- max –максимальное значение
- min_element – минимальный элемент
- max_element – максимальный элемент

Работа с коллекциями

Методы перебора всех элементов коллекции и их обработки:

- count – подсчет элементов по значению
- count_if – подсчет по условию
- find – поиск по значению
- find_if – поиск по условию
- adjacent_find – поиск одинаковых соседних пар
- for_each – перебор всех элементов и применение к ним функции
- mismatch – поиск первого различающегося элемента
- equal – сравнение на равенство последовательностей
- copy - копирование
- copy_backward – обратное копирование
- swap – обмен элементов по значению
- iter_swap – обмен элементов через указатели
- swap_ranges – обмена диапазонов
- fill – заполнение последовательности значениями
- fill_n – заполнение N элементов масива значениями
- generate – вызов функции генератора для заполнения каждого элемента последовательности
- generate_n – заполнение N элементов при помощи генератора

Работа с коллекциями

Методы перебора манипуляции элементами коллекции:

- replace – замена одного значения на другое в интервале
- replace_if – замена по условию
- transform – преобразование каждого элемента последовательности
- remove – удаление элемента по значению в интервале
- remove_if – удаление элементов по условию,
- remove_copy – копирование элементов с игнорированием по значению,
- remove_copy_if – копирование с игнорированием по условию,
- unique – получение уникальной последовательности
- unique_copy – копирование только уникальных значений
- reverse – инверсия последовательности
- reverse_copy – копирование обратной последовательности
- rotate – поворот последовательности по направлению
- rotate_copy – поворот последовательности с копированием
- random_shuffle – перемешивание последовательности
- partition – разбиение на диапазоны по условию

А где сигнатуры?

Приводить сигнатуры для каждого из перечисленных методов слишком расточительно по месту на слайдах, поэтому, просто пользуйтесь сайтом cppreference: https://en.cppreference.com/

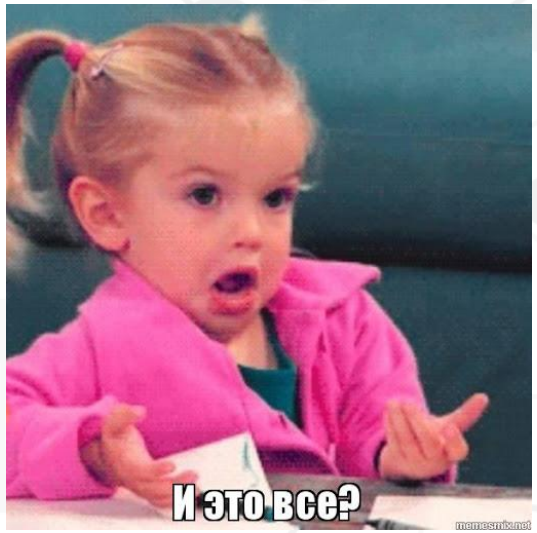
std::sort

Defined in header <algorithm>

```
template< class RandomIt >
void sort( RandomIt first, RandomIt last );                                   (1)  (until C++20)
template< class RandomIt >
constexpr void sort( RandomIt first, RandomIt last );                              (since C++20)

template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy,                                          (2)  (since C++17)
           RandomIt first, RandomIt last );

template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );                     (3)  (until C++20)
template< class RandomIt, class Compare >
constexpr void sort( RandomIt first, RandomIt last, Compare comp );                (since C++20)

template< class ExecutionPolicy, class RandomIt, class Compare >
void sort( ExecutionPolicy&& policy,                                          (4)  (since C++17)
           RandomIt first, RandomIt last, Compare comp );
```

На этом сайте есть все элементы стандартной библиотеки, от контейнеров и алгоритмов, до библиотеки параллельных потоков. Практически на каждой странице есть пример, описание искомого метода или класса, а так же его ограничения и особенности.

Пример




```
int main(){
    vector<int> v(10);
    for(auto& el: v){
        el = -50 + rand() % 100;
        cout << el << ", ";
    }
    cout << endl;
    sort(v.begin(), v.end());
    for(auto el: v){
        cout << el << ", ";
    }
    cout << endl;
}
```

```
-9, 17, -16, -50, 19, -26, 28, 8, 12, 14,
-50, -26, -16, -9, 8, 12, 14, 17, 19, 28,
```

Работа с пользовательским типом


```
class MyInt{
    private:
        int value = 0;
    public:
        int getValue(){
            return value;
        }
        void operator=(int v){
            value = v;
        }
    friend ostream& operator<<(ostream& os, MyInt& v);
};
ostream& operator<<(ostream& os, MyInt& v){
    return os << v.value;
}
```

```
int main(){
    vector<MyInt> v(10);
    for(auto& el: v){
        el = -50 + rand() % 100;
        cout << el << ", ";
    }
    cout << endl;
    sort(v.begin(), v.end());
    for(auto el: v){
        cout << el << ", ";
    }
    cout << endl;
}
```

```
D:/5-Programs/MinGW64_8.1/mingw64/lib/gcc/x86_64-w64-mingw32/8.1.0/include/c++/bits/predefined_ops.h:65:22: note:   'MyInt' is not derived from 'const __gnu_cxx::__normal_iterator<_Iterator, _Container>'
  { return *__it < __val; }
```

Предикат

Предикат — функция, возвращающая bool, также это может быть функтор оператор () которого возвращает bool. Унарный предикат — предикат принимающий 1 аргумент, к примеру !a. Бинарный предикат — предикат, принимающий 2 аргумента, примеры: a>b, a<b и др.

```
bool compareMyInt(MyInt& a, MyInt& b){
    return a.getValue() > b.getValue();
}

int main(){
    vector<MyInt> v(10);
    for(auto& el: v){
        el = -50 + rand() % 100;
        cout << el << ", ";
    }
    cout << endl;
    sort(v.begin(), v.end(), compareMyInt);
    for(auto el: v){
        cout << el << ", ";
    }
    cout << endl;
}
```

-9, 17, -16, -50, 19, -26, 28, 8, 12, 14,
28, 19, 17, 14, 12, 8, -9, -16, -26, -50,

-9, 17, -16, -50, 19, -26, 28, 8, 12, 14,
0
0
1
2
2
3
3
4
5
6
5
6
7
8
9
10
11
7
8
9
10
11
12
13
8
9
10
11
12
13
14
15
28, 19, 17, 14, 12, 8, -9, -16, -26, -50,
0

Функтор

Функторы в C++ являются сокращением от "**функциональные объекты**". **Функциональный объект** является экземпляром класса C++, в котором определён operator(). Если вы определите operator() для C++ класса, то вы получите объект, который действует как функция, но может также хранить состояние.

```
class CompareMyIntFunctor{
        int cals = 0;
    public:
        bool operator()(MyInt& a, MyInt& b){
            cout << cals << endl;
            cals++;
            return a.getValue() > b.getValue();
        }
        int getCals(){
            return cals;
        }
};
```


```
int main(){
    vector<MyInt> v(10);
    for(auto& el: v){
        el = -50 + rand() % 100;
        cout << el << ", ";
    }
    cout << endl;
    CompareMyIntFunctor functor;
    cout << functor.getCals() << endl;
    sort(v.begin(), v.end(), functor);
    for(auto el: v){
        cout << el << ", ";
    }
    cout << endl;
    cout << functor.getCals() << endl;
}
```

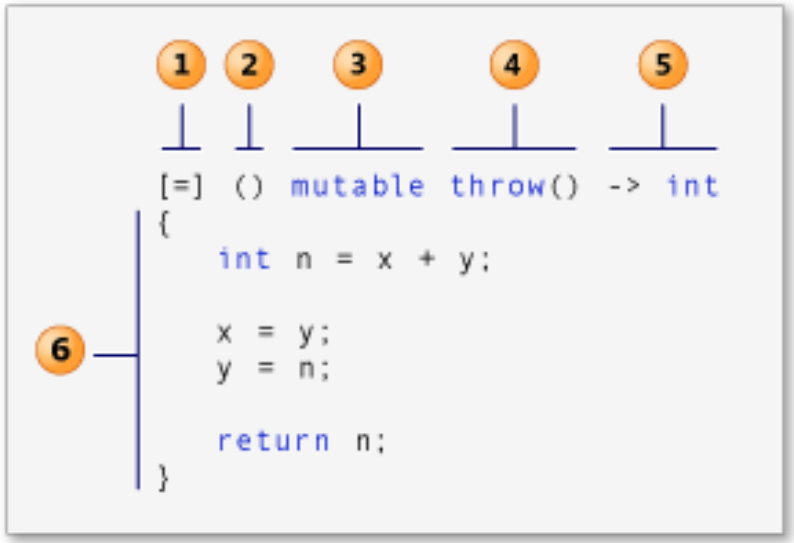
Лямбда

```
int main(){
    vector<MyInt> v(10);
    for(auto& el: v){
        el = -50 + rand() % 100;
        cout << el << ", ";
    }
    cout << endl;
    int cals = 0;
    cout << cals << endl;
    sort(v.begin(), v.end(), [&cals](MyInt& a, MyInt&b){
        cals++;
        return a.getValue() > b.getValue();
    });
    for(auto el: v){
        cout << el << ", ";
    }
    cout << endl;
    cout << cals << endl;
}
```

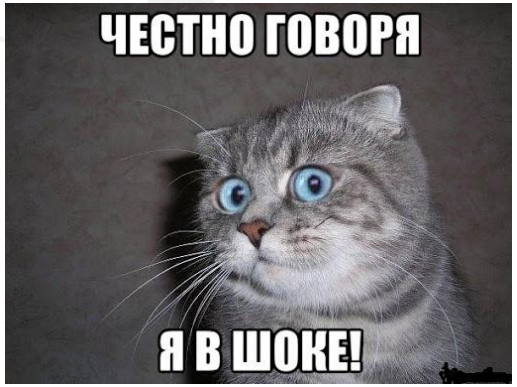
-9, 17, -16, -50, 19, -26, 28, 8, 12, 14,
0
28, 19, 17, 14, 12, 8, -9, -16, -26, -50,
31

Сигнатура лямбды

Лямбда выражения – техника программирования, сочетающая в себе преимущества указателей на функции и функциональных объектов, позволяет избежать неудобств. Как и функциональные объекты, лямбда выражения позволяют хранить состояния, но их компактный синтаксис в отличие от функциональных объектов не требует объявления класса. Лямбда выражения позволяют вам писать более компактный код и избежать ошибок, нежели используя функциональные объекты.



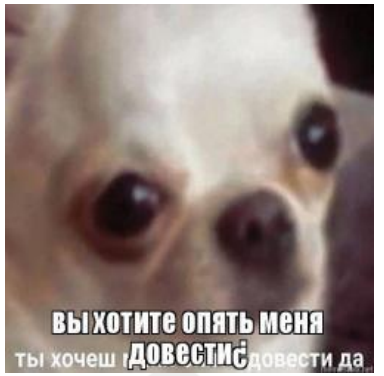
1) Маска переменных
2) Список параметров
3) Изменение параметра, переданного по значению
4) Спецификация исключения
5) Возвращаемый тип
6) Тело лямбда выражения



Части сгнатуры

Маска переменных

Лямбда выражение может получать доступ практически к любым переменным и использовать их внутри тела. Снимок определяет способ получения параметров телом лямбда выражения. Доступ к переменным, перед которыми стоит амперсанд (&), осуществляется по ссылке, а перед которыми нет амперсанда, соответственно по значению. Пустая маска ([]) означает, что тело выражения не имеет доступа к переменным.



- **[a,&b]** где a захвачена по значению, a b захвачена по ссылке.
- **[this]** захватывает указатель this по значению.
- **[&]** захват всех символов по ссылке
- **[=]** захват всех символов по значению
- **[]** ничего не захватывает

Список параметров

Список параметров лямбда выражения содержит список параметров для функции, у которых существуют следующим ограничения:

1. Список параметров не может содержать значения по умолчанию
2. Не может содержать неименованные параметры
3. Ограниченное число параметров

В стандарте **C++14** появилась возможность определять аргументы лямбда функций как auto, за счёт чего лямбда функцию можно передавать в качестве аргумента самой себе по ссылке.

Части сигнатуры

Изменение параметра переданного по значению
Спецификация позволяет разрешить телу лямбда выражения и заменить переменные, полученные по значению.
Спецификация исключений
Лямбда выражения может включать в себя спецификацию исключения, которая позволяет телу не создавать исключения.
Тело лямбда выражения
На тело выражения накладываются идентичные ограничения, как на блок кода или метод.
Возвращаемый тип
Спецификация определяет возвращаемый тип лямбда выражения. Вы можете не использовать данную конструкцию, если тело выражения имеет только одну инструкцию “**return**” или не возвращает значения вообще. Если тело содержит только одну “**return**” инструкцию, компилятор установит возвращаемый тип лямбда выражения идентичный типу “**return**” инструкции.
Это означает, что лямбда функция может использовать не только переменные, которые передаются ей в качестве параметров, но и какие-либо объекты, которые были объявлены вне **лямда-функции**.

Дополнительные подробности

Лямбда функция создаёт безымянный временный объект уникального безымянного non-union, non-aggregate типа, известного как *тип замыкания.* Благодаря введению оператора **auto** в современном стандарте C++ можно объявить объект лямбда функции довольно легко, без прописывания объявления функтора (**std::function**) со всеми параметрами и возвращаемыми значениями, что делает код более простым и читаемым

Возможные варианты синтаксиса лямбда функций

- [capture] (params) mutable exception attribute -> ret { body }
- [capture] (params) -> ret { body }
- [capture] (params) { body }
- [capture] { body }

Не имеет смысла объявлять функцию или метод в классе, если эта функция используется в одном единственном месте кода. Это как минимум будет усложнять интерфейс класса, если на каждый чих будем писать новые методы. Гораздо лучше объявить лямбду внутри другого метода, где она должна выполняться. Это касается всех лямбда функций, как обычных, так и рекурсивных.

Пример

```
int main(){
    srand(time(0));
    vector<int> v(100);
    for(auto& el: v){
        el = -50 + rand() % 100;
    }
    int positives = count_if(v.begin(), v.end(), [](int& el){ return el > 0; });
    int negatives = v.size() - positives;
    cout << positives << " vs " << negatives << endl;
    for_each(v.begin(), v.end(),
        [positives, negatives] (auto& el) mutable{
            // cout << el << " " << positives << " " << negatives << endl;
            if(el > 0 && positives > negatives){
                el *= -1;
                positives--;
                negatives++;
            }
            if(el < 0 && positives < negatives){
                el *= -1;
                positives++;
                negatives--;
            }
        }
    );
    cout << positives << " vs " << negatives << endl;
    positives = count_if(v.begin(), v.end(), [](int& el){ return el > 0; });
    negatives = v.size() - positives;
    cout << positives << " vs " << negatives << endl;
}
```

```
62 vs 38
62 vs 38
50 vs 50
```

Генерация случайных числе

Мы привыкли использовать генератор случайных числе из C, так как он прост и понятен, нам он давно известен. Однако, он имеет ряд ограничений:

- Он генерирует только целочисленные значения (перевод выполняется через деление на максимально возможное сгенерированное значение).
- Он использует только один алгоритм генерации, предоставляющий равномерное распределение чисел по всему диапазону.

Для ряда задач эти ограничения могут быть критичными, по этой причине, в стандартной библиотеке C++ **random** есть свои собственные способы генерации.

Генерация в C++ делится на 3 части:

1. Выбор основы для генератора случайных чисел – это зерно генератора, на основании которого будет строится последовательность. Зерном является **random_device** детерминированный генератор целочисленных значений, точнее результат его работы
2. Выбор алгоритма генерации, библиотека предоставляет множество вариантов генерации, один из самых популярных **mt19937** (чаще всего встречается в примерах на C++).
3. Выбор распределения, стандартное равномерное распределение **uniform_int_distribution** (для целых чисел) и **uniform_real_distribution** (для рациональных).

Пример

```
void print(vector<int>& v){
    const int VN = v.size();
    int i = 0;
    cout << "vector[" << VN << "]{";
    for_each(
        v.begin(),
        v.end(),
        [&i, &VN](auto& el){
            cout << el << ((i++ != VN - 1) ? ", ": "}");
        }
    );
    cout << endl;
}
void print(list<double>& l){
    const int LN = l.size();
    int i = 0;
    cout << "list[" << LN << "]{";
    for_each(
        l.begin(),
        l.end(),
        [&i, &LN](auto& el){
            cout << setprecision(3) << el
                 << ((i++ != LN - 1) ? ", ": "}");
        }
    );
    cout << endl
```

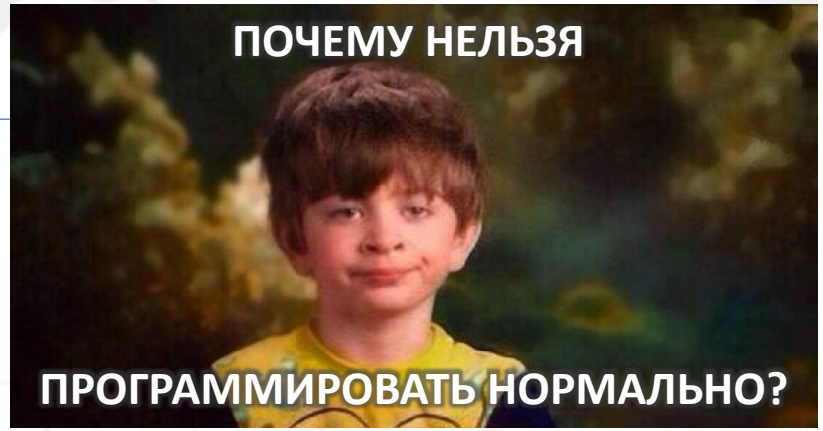
```
const int N = 20;

int main(){
    vector<int> v;
    list<double> l;
    random_device r;
    mt19937 engine(r());
    uniform_int_distribution<int> ind(-100, 100);
    uniform_real_distribution<double> dnd(-1., 1.);
    for(int i = 0; i < N; i++){
        v.push_back(ind(engine));
        l.push_back(dnd(engine));
    }
    print(v);
    print(l);
    return 0;
}
```

vector[20]{12, 83, 75, 96, -37, 58, 15, 17, -66, -97, -4, -32, 69, 30, 70, -43, -100, -9, 87, 44}
list[20]{-0.303, 0.189, 0.918, 0.323, 0.403, -0.852, -0.324, -0.938, 0.932, 0.559, -0.868, -0.17, -0.434, 0.141, 0.121, 0.459, -0.476, -0.553, -0.0438, 0.771}

Унифицируем пример

```
template <typename Type, class Container>
Container generate(
    const int& N,
    const Type& left = std::is_floating_point<Type>::value ? -1.: -100,
    const Type& right = std::is_floating_point<Type>::value ? 1.: 100
){
    std::random_device r;
    std::mt19937 engine(r());
    Container container;
    if constexpr (std::is_floating_point<Type>::value){
        std::uniform_real_distribution<Type> dist(left, right);
        for(int i = 0; i < N; i++) container.push_back(dist(engine));
    } else {
        std::uniform_int_distribution<Type> dist(left, right);
        for(int i = 0; i < N; i++) container.push_back(dist(engine));
    }
    return container;
}
```

Унифицируем пример




```
template <class Container>
void print(const Container& v){
    const int VN = v.size();
    int i = 0;
    std::string name = __PRETTY_FUNCTION__;
    name = name.substr(name.find_last_of("::") + 1, name.find("]") - 1);
    std::cout << name <<"[" << VN << "]{";
    std::for_each(
        v.begin(),
        v.end(),
        [&i, &VN](auto& el){
            std::cout << el << ((i++ != VN - 1) ? ", ": "}");
        }
    );
    std::cout << std::endl;
}
```

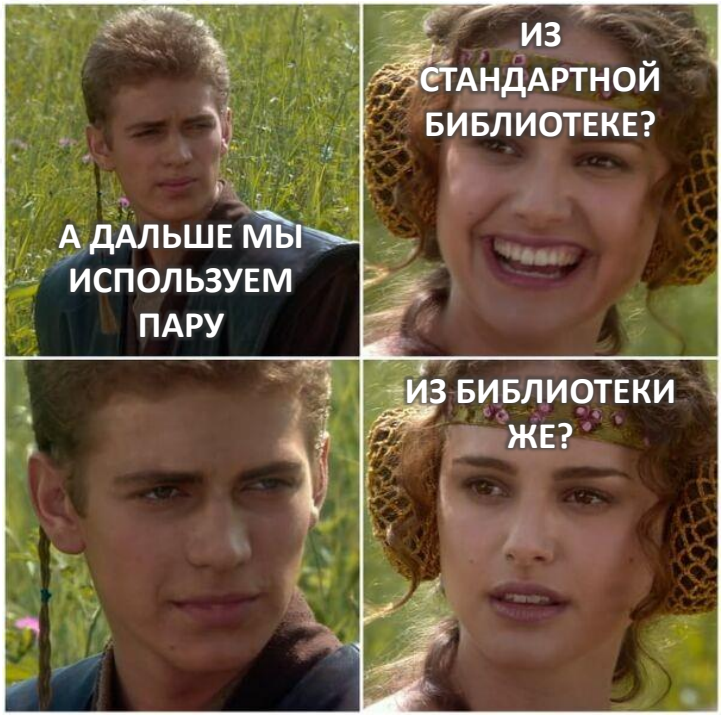
Унифицируем пример

```
#include <vector>
#include <list>
#include <set>
#include <deque>

#include "base.h"

using namespace std;

const int N = 20;

int main(){
    vector<int> v = generate<int, vector<int>>(N);
    list<double> l = generate<double, list<double>>(N);
    deque<long long int> d = generate<long long int, deque<long long int>>(N, -
1000000000000, 1000000000000);

    print(v);
    print(l);
    print(d);
    return 0;
}
```

vector<int>][20]{12, 45, -30, 83, -99, 19, 75, 3, 92, 96, -66, 32, -37, 6, 40, 58, -49, -86, 15, -34}
list<double>][20]{0.452498, 0.830679, 0.188648, 0.0307165, 0.950299, 0.323121, 0.0573049, 0.576986, -0.851799, -0.3403, 0.167488, -0.938056, 0.857187, -0.960495, 0.558744, -0.32791, -0.314948, -0.170361, 0.814626, 0.302961}
deque<long long int>][20]{124105681002, -300283945447, -988857462583, 745970169002, 919743648206, -657856202574, -366369262418, 403545890237, -492875599044, 148172963230, -323182973939, -523125605698, -652414337696, 932820119257, 309017435041, -45073958523, -869679843303, -972448559986, 687524060525, -434561197696}

Пример с сортировками

```
template <typename FT, typename ST>
class Pair:public pair<FT, ST>{
    public:
        Pair(const FT& a, const ST& b): pair<FT, ST>(a, b) {};
        template <typename FFT, typename FST>
        friend ostream& operator<<(ostream& out, const Pair<FFT, FST>& value);
};

template <typename FFT, typename FST>
ostream& operator<<(ostream& out, const Pair<FFT, FST>& value){
    return out << "(" << value.first << "," << value.second << ")";
}

bool equal_order(vector<Pair<int, int>> a, vector<Pair<int, int>> b){
    return std::equal(
        a.begin(), a.end(), b.begin(),
        [](const Pair<int, int>&a, const Pair<int, int>&b){
            return a.first == b.first && a.second == b.second;
        }
    );
}
```



Пример с сортировками

```
int main(){
    vector<int> v = generate<int, vector<int>>(N, 0, 10);
    vector<Pair<int, int>> v_pair;
    int i = 0;
    for(int el: v) v_pair.push_back(Pair<int, int>(el, i++));
    vector<int> v_copy = v;
    vector<Pair<int, int>> v_pair_copy = v_pair;
    print(v_copy);
    sort(v_copy.begin(), v_copy.end());
    print(v_copy);
    print(v_pair_copy);
    sort(v_pair_copy.begin(), v_pair_copy.end(), [](const Pair<int, int>& a, const Pair<int, int>& b){return a.first < b.first;});
    print(v_pair_copy);
    …
    cout << endl;
    vector<Pair<int, int>> v_pair_t_copy = v_pair;
    print(v_pair_t_copy);
    stable_sort(v_pair_t_copy.begin(), v_pair_t_copy.end(), [](const Pair<int, int>& a, const Pair<int, int>& b){return a.first <
b.first;});
    print(v_pair_t_copy);

    cout << "equal order first(sort), third(stable_sort): " << equal_order(v_pair_copy, v_pair_t_copy) << endl;
    …
    return 0;
}
```

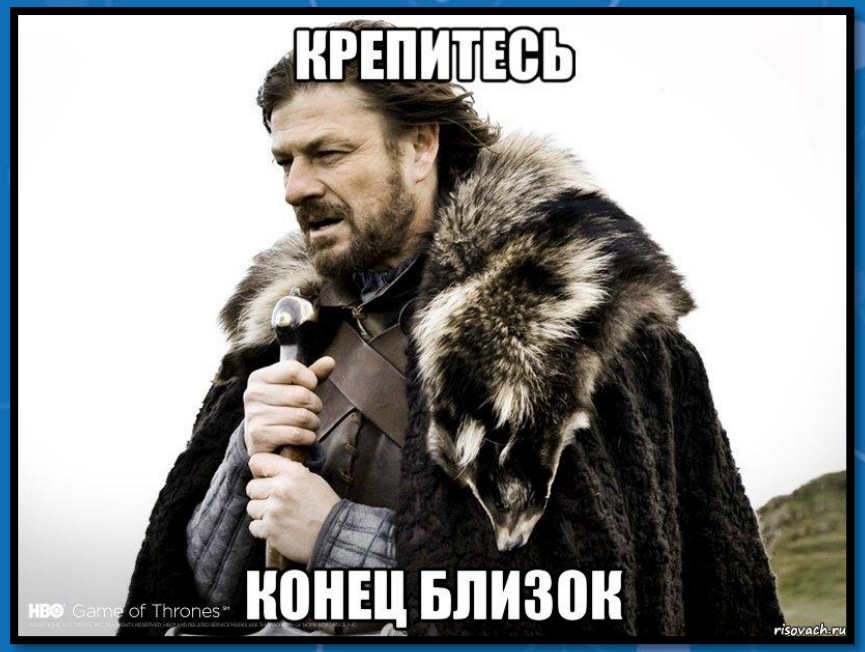
Пример с сортировками

vector<int>][20]{6, 6, 0, 8, 7, 9, 10, 5, 1, 6, 9, 4, 8, 7, 1, 2, 5, 0, 1, 0}
vector<int>][20]{0, 0, 0, 1, 1, 1, 2, 4, 5, 5, 6, 6, 6, 7, 7, 8, 8, 9, 9, 10}
vector<Pair<int, int> >][20]{(6,0), (6,1), (0,2), (8,3), (7,4), (9,5), (10,6), (5,7), (1,8), (6,9), (9,10), (4,11), (8,12), (7,13), (1,14), (2,15), (5,16), (0,17), (1,18), (0,19)}
vector<Pair<int, int> >][20]{(0,19), (0,2), (0,17), (1,18), (1,8), (1,14), (2,15), (4,11), (5,16), (5,7), (6,1), (6,9), (6,0), (7,13), (7,4), (8,12), (8,3), (9,10), (9,5), (10,6)}

vector<Pair<int, int> >][20]{(6,0), (6,1), (0,2), (8,3), (7,4), (9,5), (10,6), (5,7), (1,8), (6,9), (9,10), (4,11), (8,12), (7,13), (1,14), (2,15), (5,16), (0,17), (1,18), (0,19)}
vector<Pair<int, int> >][20]{(0,19), (0,2), (0,17), (1,18), (1,8), (1,14), (2,15), (4,11), (5,16), (5,7), (6,1), (6,9), (6,0), (7,13), (7,4), (8,12), (8,3), (9,10), (9,5), (10,6)}
equal order first(sort), second(sort): 1

vector<Pair<int, int> >][20]{(6,0), (6,1), (0,2), (8,3), (7,4), (9,5), (10,6), (5,7), (1,8), (6,9), (9,10), (4,11), (8,12), (7,13), (1,14), (2,15), (5,16), (0,17), (1,18), (0,19)}
vector<Pair<int, int> >][20]{(0,2), (0,17), (0,19), (1,8), (1,14), (1,18), (2,15), (4,11), (5,7), (5,16), (6,0), (6,1), (6,9), (7,4), (7,13), (8,3), (8,12), (9,5), (9,10), (10,6)}
equal order first(sort), third(stable_sort): 0

vector<Pair<int, int> >][20]{(6,0), (6,1), (0,2), (8,3), (7,4), (9,5), (10,6), (5,7), (1,8), (6,9), (9,10), (4,11), (8,12), (7,13), (1,14), (2,15), (5,16), (0,17), (1,18), (0,19)}
vector<Pair<int, int> >][20]{(0,2), (0,17), (0,19), (1,8), (1,14), (1,18), (2,15), (4,11), (5,7), (5,16), (6,0), (6,1), (6,9), (7,4), (7,13), (8,3), (8,12), (9,5), (9,10), (10,6)}
equal order third(stable_sort), fourth(stable_sort): 1



Спасибо за внимание