

Шаблонные функции с контейнером на вход

-> написать шаблонную функцию, которая ищет макс значение в константном или неконстантном контейнере

```
#include <iostream>
#include <list>
#include <vector>

using namespace std;

template <typename T>
typename T::value_type Max(const T& c) {
    // используем константные итераторы cbegin
    typename T::value_type max = *c.cbegin();

    for (const auto& it : c) {
        if (*it > max) {
            max = *it;
        }
        max = *it > max ? *it : max
    }

    return max;
}

int main() {
    list<int> l = { 1, 2, 3, 4, 5 };
    cout << Max<list<int>>(l) << endl;

    vector<long> v = { 1, 2, 3, 4, 5 };
    cout << Max<vector<long>>(v) << endl;

    return 0;
}
```

Шаблонная функция, которая удваивает все элементы в контейнере Из [1, 2, 3] делает [1, 1, 2, 2, 3, 3]

```
// работает, но херня, много памяти жрёт
template <typename T>
void dbl(T& c) {
    T res;
    auto it = c.begin();
    while (it != c.end()) {
        res.push_back(*it);
        res.push_back(*it);
        ++it;
    }
}
```

```

    }

    c = res;

    return;
}

// работает, всё правильно
template <typename T>
void dbl(T& c) {
    auto it = c.begin();
    // typename T::iterator it = c.begin(); - порой могут вот так просить
на коллоквиуме

    while (it != c.end()) {
        it = c.insert(++it, *it);
        ++it;
    }

    return;
}

```

вход функции: 2 итератора, реверсирующая данную коллекцию Было [1, 2, 3] Стало [3, 2, 1]

```

template <typename T>
void rever(T b, T e) {
    if (b == e) return; // для проверки на пустой контейнер

    --e;

    while (b != e) { // просто сравнивать нельзя у листов
        swap(*b++, *e); // нативный swap

        if (b == e) break;

        --e;
    }

    return;
}

```

функции подаётся 2 итератора, предикат и стандартное значение Предикат - объект с перегруженными круглыми скобками

```

template<typename T, typename P> // T - указатель P - предикат
void f(T b, T, e, P pred, typename T::value_type val) {

}

```

```
// пример предиката
struct Pred {
    bool operation()(int v) {
        return v % 3;
    }
}

int main () {
    vector<int> v = {...};

    f(v.begin(), v.end(), Pred(), 9);

    return 0;
}
```

Лямба-функции

Неименованные функции, которые можно просто вставить в выражение

```
int x = 5;
auto ef = [x]() { cout << x; } // тут не обойтись без auto
ef();
auto ef = [](int a) { cout << a; }
ef(x);
```

В квадратных скобках - привязка к внешним элементам

- [] привязки нет
- [=] привязываются все внешние
- [&] по ссылке привязываются все внешние

В круглых - к тем, что передаются

Алгоритм remove_if

```
vector<int> v = {2, 4, 5, 7, 9, 10};
auto end = remove_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; })
```

remove_if перемещает в хвост все значения, для которых предикат (лямбда-функция) верна. Возвращает указатель на начало хвоста Вывод: [5, 7, 9, 2, 4, 10]. end указывать будет на '2'

2 задание 3го контекста

есть уже complex, complex stack. Добавление в стек через операцию <<

посчитать польско-инвер запись для комплексных чисел. Если попадается z - значение по умолчанию

map - словарь. Содержит элементы 'ключ' - 'значение'

```
complex eval(const vector<string>& args, const complex& z) {
    complex stack st;
    map<string, function<void()>> mp; {
        {"z", [&st, &z]() { st = st << z; }}, // если попало z, добавили
В стек
        {";", [&st]() { st = ~st; }},
        ... // все прочие +/-/... операции
    }

    for (const auto& iter : args) {
        if (s[0] != '(') { // если строка не (re,im)
            mp[s](); // просто вызвали нужное
        } else {
            st = st << complex(s);
        }
    }

    // ответ на верхушке стека
    return +st;
}
```