**Алгоритмы**

#include <algorithm>

В STL более 100 алгоритмов. Мы будем рассматривать не все.
Мы не будем рассматривать:

* Операции с неинициализированными данными.
* Алгоритмы для работы с кучей.
* Операции над множествами.
* Операции с перестановками.

Мы будем рассматривать следующие алгоритмы:

* Микро-алгоритмы
* Алгоритмы, не модифицирующие последовательности
* Алгоритмы типа find
* Модифицирующие алгоритмы
1. **Микро-алгоритмы**
	* swap(T &a, T &b)
	Меняет местами значения двух элементов.
	* iter\_swap(It p, It q)
	Меняет местами значения элементов, на которые указывают итераторы.
	* max(const T &a,const T &b )
	Возвращает максимальный элемент.
	* min(const T &a,const T &b )
	Возвращает минимальный элемент.

У этих алгоритмов есть версии с тремя параметрами. Третий параметр принимает бинарный предикат, задающий упорядоченность объектов.

1. **Алгоритмы, не модифицирующие последовательности**
	* size\_t count(It p, It q, const T &x)
	Возвращает, сколько раз элемент со значением x входит в последовательность, заданную итераторами p и q.
	* size\_t count\_if(It p, It q, Pr pred)
	Возвращает, сколько раз предикат pred возвращает значение true.
	Например, count\_if(p, q, divides\_by(8)) вернет, сколько элементов кратно 8;
2. **Алгоритмы типа find**
	* find(It p, It q, const T &x)
	Возвращает итератор на первое вхождение элемента x в последовательность, заданную итераторами p и q.
	* find\_if(It p, It q, Pr pred)
	Возвращает итератор на первый элемент, для которого предикат pred вернул значение true.
	* find\_first\_of(It p, It q, Itr i, Itr j)
	Возвращает итератор на первое вхождение любого элемента из последовательности, заданной итераторами i и j, в последовательность, заданную итераторами p и q. Последовательности могут быть разных типов (например std::vector и std::list).
	* min\_element(It p, It q)
	Возвращает итератор на минимальный элемент последовательности.
	* max\_element(It p, It q)
	Возвращает итератор на максимальный элемент последовательности.
	* equal(It p, It q, Itr i)
	Сравнивает две последовательности на эквивалентность. Вторая последовательность задается одним итератором, так как последовательности должны быть одинаковой длины. Если вторая короче, то undefined behaviour.
	* pair <It, Itr> mismach(It p, It q, Itr i)
	Возвращает пару итераторов, указывающую на первое несовпадение последовательностей.
	* F for\_each(It p, It q, F func)
	Для каждого элемента последовательности применяет функтор func. Возвращаемое значение функтора после каждого применения игнорируется. Возвращает функтор func после его применения ко всем элементам.
	* bool binary\_search(It p, It q, const T &x)
	Возвращает true, если в упорядоченной последовательности есть элемент, значение которого равно x, false в противном случае.
	Если хотим получить итератор на элемент со значением x, то нужно использовать алгоритмы lower\_bound(It p, It q, const T &x),upper\_bound(It p, It q, const T &x), equal\_range(It p, It q, const T &x), которые выполняют то же, что и одноименные методы для контейнера std::set. Эти алгоритмы работают за линейное время на BiDi итераторах и за логарифмическое время на RA итераторах.

Все эти алгоритмы имеют версии с параметром, принимающим бинарный предикат, задающий упорядоченность объектов.

1. **Модифицирующие алгоритмы**
	* fill(It p, It q, const T &x), fill\_n(It p, Size n, const T &x)
	Заполняют последовательность значениями, равными значению x.
	* generate(It p, It q, F gen), generate\_n(It p, Size n, F gen)
	Заполняют последовательность значениями, сгенерированными функтором gen (например, генератором случайных чисел).
	* random\_shuffle(It p, It q), random\_shuffle(It p, It q, F &rand)
	Перемешивает элементы в случайном порядке: меняет местами каждый элемент с элементом, номер которого выбирается случайно. Третьим параметром можно задать функтор, который будет выбирать этот случайный номер. Можно передавать генератор случайных чисел, но распределение должно быть равномерным (каждая перестановка должна генерироваться с вероятностью 1/n!, а это совсем не то же самое, что каждый элемент окажется на i-м месте с вероятностью 1/n). Требует RA итераторов.
	* copy(It p, It q, Itr out)
	Копирует значения элементов последовательности, заданной итераторами p и q, в последовательность, начинающуюся с итератора out.
	copy\_backward(It p, It q, Itr out) Копирует элементы последовательности, заданной итераторами p и q, в последовательность, заканчивающуюся итератором out. Итераторы должны быть BiDi.
	* remove\_copy(It p, It q, Itr out, const T &x)
	Копирует значения элементов из последовательности, заданной итераторами p и q, в последовательность, начинающуюся с итератора out, за исключением элементов, значения которых равны значению x.
	remove\_copy\_if(It p, It q, Itr out, Pr pred)
	Копирует значения элементов из последовательности, заданной итераторами p и q, в последовательность, начинающуюся с итератора out, за исключением элементов, для которых предикат pred возвращает значение true.
	* reverse(It p, It q)
	Переставляет элементы в обратном порядке.
	reverse\_copy(It p, It q, Itr out)
	Копирует значения элементов в обратном порядке.
	* rotate(It p, It middle, It q)
	Сдвигает элементы последовательности так, что элемент, на который указывает итератор middle становится первым.
	
	Этот алгоритм реализован по-разному для разных категорий итераторов (RA, BiDi, Fwd).
	* swap\_ranges(It p, It q, Itr i)
	Меняет местами элементы последовательности, заданной итераторами p и q, с соответствующими элементами последовательности, начинающейся с и итератора out.
	
	* remove(It p, It q, const T &x)
	Удаляет из последовательности элементы, значения которых совпадают по значению с x. Возвращает итератор на новый конец последовательности.
	Например:
	* using namespace std;
	* vector<int> v;
	* ...
	* vector<int>::iterator new\_end = std::remove(v.begin(), v.end(), 3); // хотим удалить все тройки

В действительности remove ничего не удаляет, так как ему не передается контейнер.

remove внутри себя как бы вызывает remove\_copy\_if. Поэтому последние два элемента в примере на рисунке останутся без изменений.
Правильный вариант удаления:

v.erase(std::remove(v.begin(), v.end(), 3), v.end());

Удаляем все, что находится между итератором, который вернул remove, и концом последовательности. Это называется remove-erase-idiom.
remove\_if(It p, It q, Pr pred)
Удаляет из последовательности элементы, для которых предикат pred возвращает true. Возвращает итератор на новый конец последовательности.

* + unique(It p, It q), unique(It p, It q, Pr pred)
	Удаляет одинаковые подряд идущие элементы, оставляя только по одному элементу для каждого значения. Элементы последовательности должны быть отсортированы. Работает аналогично алгоритмам remove и remove\_if, оставляя в начале только уникальные элементы, а в конце - то, что осталось. В качестве третьего параметра можно передавать предикат, сравнивающий два элемента и возвращающий true, если элементы равны, иfalse в противном случае.
	unique\_copy(It p, It q, Itr out), unique\_copy(It p, It q, Itr out, Pr pred)
	Копирует уникальные элементы в последовательность, начинающуюся с итератора out.
	+ transform(It p, It q, Itr out, F func)
	К каждому элементу входящей последовательности применяет функтор func и записывает результат в последовательность, начинающуюся с итератора out.
	transform(It p, It q, Itr i, Iter out, F func)
	Применяет бинарный функтор func к каждой паре элементов из двух входящих последовательностей и записывает результат в результирующую последовательность.
	+ accimulate(It p, It q, T i, F func)
	Последовательно применяет бинарный функтор func к парам (i, \*p++), где i - некоторое начальное значение, которое затем каждый раз заменяется значением, которое возвращает функтор. Функтор должен возвращать значение типа T.
	Реализация этого алгоритма выглядит примерно следующим образом:
	+ while (p != q)
	+ {
	+ i = func(i, \*(p++));
	+ }
	+ return i;

Например, если в качестве i передать 0, а в качестве func - функтор, вычисляющий сумму, то посчитаем сумму элементов последовательности. Если в качестве i передать 1, а в качестве func - функтор, вычисляющий произведение, то получим произведение элементов и т.д.

* + sort(It p, It q), sort(It p, It q, Pr pred)
	Сортирует элементы последовательности в порядке возрастания. stable\_sort(It p, It q), stable\_sort(It p, It q, Pr pred)
	Сортирует элементы, сохраняя порядок элементов с одинаковыми значениями относительно друг друга. Эти алгоритмы требуют RA итераторов, поэтому на списке работать не будут. Но у списка есть собственные функции члены sort, stable\_sort.
	+ void nth\_element(It p, It nth, It q), void nth\_element(It p, It q, It nth, Pr pred)
	Позволяет получить n-й по порядку элемент (n-й по счету, как если бы массив был отсортирован), переставляя элементы таким образом, что все элементы до него меньше, либо равны ему, а элементы после - больше, либо равны ему.
	+ partition(It p, It q, Pr pred)
	Переставляет элементы последовательности таким образом, что все элементы, для которых предикат вернул true, предшествуют тем, для которых он вернул false. Возвращает итератор на первый элемент из второй группы.
	+ void partial\_sort(It p, It middle, It q), void partial\_sort(It p, It middle, It q, Pr pred)
	Переставляет элементы последовательности так, что элементы межу итераторами p и q располагаются в том порядке, как если бы последовательность была отсортирована, а элементы в оставшейся части - в произвольном порядке. То есть получаем часть отсортированной последовательности (не то же самое, что отсортированную часть).
	+ merge(It p, It q, Itr i, Itr j, Iter out), merge(It p, It q, Itr i, Itr j, Iter out, Pr pred)
	Сортирует две последовательности слиянием.
1. При вызове remove\_copy мы не знаем, сколько элементов скопируется (сколько элементов выкинем).
Например:
2. using namespace std;
3. vector<int> v;
4. vector<int> w;
5. // хотим скопировать элементы из v в w без пятерок
6. w.resize(v.size());
7. vector<int>:: iterator it = remove\_copy(v.begin(), v.end(), w.begin(), 5);
8. w.erase(it, w.end());

Однако мы выполнили лишние действия, можно сделать лучше. Правильный подход:

using namespace std;

vector<int> v;

vector<int> w;

// хотим скопировать элементы из v в w без пятерок

remove\_copy(v.begin(), v.end(), back\_inserter(w), 5);

back\_inserter - конструкция, которая добавляет элементы в конец контейнера (как бы делает push\_back). Эта конструкция работает для всех контейнеров. Таким образом, в нашем примере нам не нужно делать resize и erase.
Кроме back\_inserter есть front\_inserter. Это функции, которые возвращают итератор (на конец и на начало контейнера соответственно).

1. Итераторы потоков ввода-вывода:
	* istream\_iterator
	* ostream\_iterator

Быстрый способ скопировать содержимое потока:

using namespace std;

copy(istream\_iterator<double>(f), istream\_iterator<double>(), ...);

Или, например:

using namespace std;

vector<double> v(*(*istream\_iterator<double>(f)*)*, istream\_iterator<double>());

...

v.clear();

Без скобок, выделенных курсивом, не скомпилируется, так как это похоже на объявление функции, а не на вызов конструктора от временных объектов:

A a( B() ); // это объявление функции

A a(( B() )); // а это - вызов конструктора копирования от временного объекта

Быстрый способ вывести содержимое контейнера на поток:

using namespace std;

vector<int> v;

copy(v.begin(), v.end(), ostream\_iterator(cout, " "));

Второй параметр - разделитель. Выведет на стандартный поток вывода все элементы вектора, разделяя пробелами.