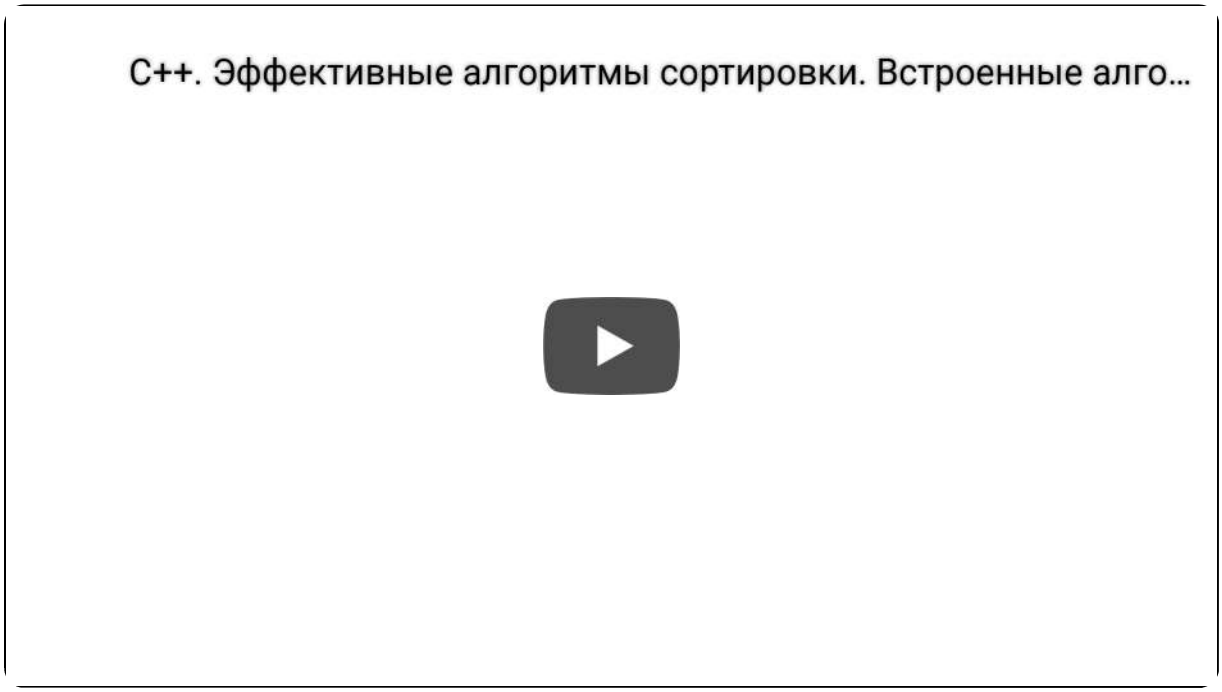
Встроенные алгоритмы сортировки



В языке есть C++ есть встроенная функция `sort` для сортировки массива.

Рассмотрим пример её применения:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> a{5, 2, 3, 1, 4};
    sort(a.begin(), a.end());
    for (int i = 0; i < a.size(); ++i)
        cout << a[i] << " ";
    return 0;
}
```

Обратите внимание, что для использования функции `sort` необходимо подключить библиотеку `algorithm`.

Стандартная функция сортировки имеет сложность — $O(n \log n)$

Аргументами функции `sort(it1, it2, comp)` являются:

- `it1` — итератор на начало сортируемого отрезка массива;
- `it2` — итератор на следующий за последним элементом сортируемого отрезка массива;
- `comp` (необязательный) — критерий упорядочивания массива (функция-компаратор).

Для вектора существуют специальные итераторы:

- `begin()` — итератор, указывающий на начало вектора;
- `end()` — итератор, указывающий на следующий за последним (несуществующий) элемент.

Также существуют reverse-итераторы. Reverse-итераторы — итераторы, которые двигаются в обратном порядке, то есть от конца массива к его началу. Они полезны, если необходимо отсортировать массив в порядке невозрастания.

- `rbegin()` — итератор, указывающий на последний элемент вектора.
- `rend()` — итератор, указывающий на предыдущий перед началом вектора (несуществующий) элемент.

Таким образом, чтобы отсортировать массив `a` в порядке невозрастания, необходимо написать следующую инструкцию:

```
sort(a.rbegin(), a.rend(), cmp);
```

Пусть необходимо отсортировать массив по какой-то необычной функции. Например, необходимо отсортировать массив чисел в порядке неубывания модулей его элементов.

Для решения этой задачи напишем функцию, которую мы будем использовать для сравнения элементов массива (компаратор).

```
bool cmp(int x, int y)
{
    return abs(x) < abs(y);
}
```

Функция-компаратор всегда реализует сравнение типа "меньше". То есть если `cmp(x, y)` возвращает истину, то по нашему правилу сравнения x должен стоять в отсортированном массиве раньше y.

Теперь для сортировки элементов массива по их модулю достаточно написать:

```
sort(a.begin(), a.end(), cmp);
```

Аналогичным образом можно написать компаратор для сравнения двух чисел по их последней цифре:

```
bool cmp(int x, int y)
{
    return x % 10 < y % 10;
}
```

Функцию `sort` можно использовать и для упорядочивания более сложных объектов. Например, решим задачу упорядочивания точек на плоскости по удалённости от начала координат. Для этого создадим структруру `Point` для хранения точек:

```
struct Point
{
    int x, y;
};
```

Добавим компаратор:

```
bool cmp(Point p, Point q)
{
   return p.x * p.x + p.y * p.y <q.x * q.x + q.y * q.y;
}
```

Теперь можно отсортировать массив точек:

```
vector<Point> a = {
        {1, 3},
        {2, 4},
        {1, 2},
        {2, -3}};
```

```
sort(a.begin(), a.end(), cmp);
```

Отсортируем точки по возрастанию x-координаты. Для этого напишем компаратор:

```
bool cmp(Point p, Point q)
{
   return p.x < q.x;
}
```

Заметим, что при таком компараторе нельзя ничего сказать про порядок следования точек $(1, 3)$ и $(1, 2)$. В языке C++ в этом случае функция sort упорядочивает элементы произвольным образом. Однако, в C++ есть другая функция сортировки: `stable_sort`. При вызове такой функции у элементов с одинаковым значением сохраняется порядок следования. Сортировки, обладающие таким свойством, называются **устойчивыми**.

Рассмотрим пример:

```
vector<Point> a = {
        {1, 3},
        {2, 4},
        {1, 2},
        {2, -3}};
    stable_sort(a.begin(), a.end(), cmp);
```

В итоге в массиве a точки будут записаны в следующем порядке: $(1, 3), (1, 2), (2, 4), (2, -3)$.

Можно переделать компаратор таким образом, чтобы точки при разных координатах x упорядочивались по x-координате, а при одинаковых — по y:

```
bool cmp(Point p, Point q)
{
   if (p.x != q.x)
        return p.x < q.x;
   return p.y < q.y;
}
```