

Лекція 27.

Класифікація алгоритмів модифікуючих послідовності. Remove- erase ідіома.



План на сьогодні

1

Модифікуючі алгоритми

2

Копіювання / переміщення

3

Заповнення / генерація

4

Перетворення

5

Видалення. remove-erase ідіома

6

Перестановка

7

Розбиття / упорядкування



Модифікуючі алгоритми



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Модифікуючі алгоритми

Модифікуючі алгоритми — це функції з заголовка `<algorithm>`, які змінюють вміст або порядок елементів у діапазоні через присвоєння, переміщення чи логічне видалення.

- Працюють із будь-якими контейнерами, масивами чи діапазонами, що надають не-`const` ітератори.
- Інкапсулюють поширені шаблони роботи з даними, підвищують читабельність коду.
- Гарантують відомі межі складності (переважно $O(n)$ або $O(n \log n)$), що дозволяє легко оцінювати продуктивність.

Класифікація модифікуючих алгоритмів

- Копіювання / переміщення: `copy`, `move`, `copy_if`.
- Заповнення / генерація: `fill`, `generate`, `iota`.
- Перетворення: `transform`, `replace`, `replace_if`.
- Видалення (логічне): `remove`, `remove_if`, `unique`.
- Перестановка: `rotate`, `shuffle`, `reverse`, `swap_ranges`.
- Розбиття / упорядкування: `partition`, `stable_partition`, `sort`, `stable_sort`.

Копіювання / переміщення



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Копіювання / переміщення

Метод	Короткий опис
<code>copy(InputIt first, InputIt last, OutputIt d_first)</code>	Копіює елементи з <code>[first, last)</code> у діапазон, що починається з <code>d_first</code> ; повертає ітератор за останнім скопійованим елементом.
<code>copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPredicate pred)</code>	Копіює лише ті елементи, для яких <code>pred(*it) == true</code> , зберігаючи порядок; повертає кінець вихідного діапазону.
<code>copy_n(InputIt first, Size count, OutputIt d_first)</code>	Копіює рівно <code>count</code> елементів, починаючи з <code>first</code> , у діапазон з <code>d_first</code> .
<code>copy_backward(BidirIt1 first, BidirIt1 last, BidirIt2 d_last)</code>	Копіює елементи з <code>[first, last)</code> у діапазон, що закінчується в <code>d_last</code> , рухаючись справа-наліво; корисно при перекриванні «праворуч».
<code>move(InputIt first, InputIt last, OutputIt d_first)</code>	Переміщує елементи з <code>[first, last)</code> у діапазон, що починається з <code>d_first</code> ; вихідні об'єкти залишаються в допустимому, але невизначеному стані.
<code>move_backward(BidirIt1 first, BidirIt1 last, BidirIt2 d_last)</code>	Переміщує елементи з <code>[first, last)</code> у діапазон, що закінчується в <code>d_last</code> , рухаючись справа-наліво; застосовується при перекриванні «праворуч».

copy і copy_if

`copy` копіює усі елементи з діапазону `[first, last)` до діапазону, що починається з `d_first`, не змінюючи їхнього порядку, і повертає ітератор одразу за останнім скопійованим елементом. `copy_if` діє так само, але додає фільтр: до вихідного діапазону потрапляють лише ті елементи, для яких предикат повертає `true`.

```
int main() {
    vector<int> sourceValues = { 1, 2, 3, 4 };
    vector<int> destinationValues(sourceValues.size());
    copy(sourceValues.begin(), sourceValues.end(), destinationValues.begin());
    // Копіювання елементів в потік виводу
    copy(destinationValues.begin(), destinationValues.end(), ostream_iterator<int>(cout, " ")); // 1 2 3 4
    cout << '\n';

    vector<int> numbers = { 1, 2, 3, 4, 5, 6 };
    vector<int> evenNumbers;
    copy_if(numbers.begin(), numbers.end(), back_inserter(evenNumbers),
        [](int value) { return value % 2 == 0; });

    for (int value : evenNumbers) {
        cout << value << ' '; // 2 4 6
    }
}
```


std::back_inserter

`std::back_inserter` — це ітератор-адаптер, який дозволяє вставляти елементи в кінець контейнера за допомогою таких алгоритмів, як `std::copy`, `std::transform` тощо.

Він створює об'єкт `std::back_inserter`, який використовує `container.push_back(value)` для кожного нового елемента.

```
int main() {  
    ifstream in("test.txt");  
    vector<int> destination;  
    // Копіювання елементів з потоку вводу  
    copy(istream_iterator<int>(in), istream_iterator<int>(), back_inserter(destination));  
  
    // Копіювання елементів в потік виводу  
    copy(destination.begin(), destination.end(), ostream_iterator<int>(cout, " "));  
}
```

move і move_backward

`move` переносить ресурси елементів до нового місця призначення, лишаючи вихідні об'єкти у припустимому, але невизначеному стані й не дозволяючи перекривання «праворуч».

`move_backward` виконує те саме, але йде справа-наліво, тому безпечно працює, коли діапазони накладаються з правого краю.

```
int main() {
    vector<string> originalNames = { "Ada", "Grace", "Linus" };
    vector<string> movedNames(originalNames.size());
    move(originalNames.begin(), originalNames.end(), movedNames.begin());

    copy(movedNames.begin(), movedNames.end(), ostream_iterator<string>(cout, " ")); // Ada Grace Linus
    cout << '\n';

    vector<string> sourceWords = { "zero", "one", "two" };
    vector<string> targetWords(5, "-"); // - - - - -
    move_backward(sourceWords.begin(), sourceWords.end(), targetWords.end());

    for (const string& value : targetWords) {
        cout << value << ' ';
    }
    cout << '\n'; // - - zero one two
}
```

copy_n і copy_backward

`copy_n` копіює рівно `count` елементів, починаючи з `first`, що зручно, коли довжину потрібно задати явно. `copy_backward` копіює справа-наліво, тож зберігає порядок і безпечний при перекриванні «праворуч».

```
int main() {  
    array<int, 5> values = {10, 20, 30, 40, 50};  
    vector<int> firstThree(3, 3);  
    copy_n(values.begin(), 2, firstThree.begin());  
  
    for (int value : firstThree) cout << value << ' '; cout << '\n';    // 10 20 3  
  
    vector<int> numbers = {1, 2, 3, 4, 5};  
    copy_backward(numbers.begin(), numbers.end() - 1, numbers.end());  
  
    for (int value : numbers) cout << value << ' '; cout << '\n';    // 1 1 2 3 4  
}
```

Заповнення / генерація



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Заповнення / генерація

Метод	Короткий опис
<code>fill(ForwardIt first, ForwardIt last, const T& value)</code>	Записує копії value у всі позиції діапазону [first, last).
<code>fill_n(OutputIt first, Size count, const T& value)</code>	Записує value count разів, починаючи з first; повертає ітератор одразу за останнім зміненим елементом.
<code>generate(ForwardIt first, ForwardIt last, Generator gen)</code>	Для кожної позиції викликає gen() і зберігає результат, отже діапазон заповнюється згенерованими значеннями.
<code>generate_n(OutputIt first, Size count, Generator gen)</code>	Виконує gen() рівно count разів, записуючи отримані значення; повертає ітератор кінця заповненого фрагмента.

fill i fill_n

Функція `fill` записує одну й ту саму величину до всіх елементів діапазону `[first, last)`, тоді як `fill_n` робить те саме, але рівно `count` разів, починаючи з `first`; обидві працюють за лінійний час і повертають ітератор одразу за останнім зміненим елементом.

```
int main() {  
    vector<int> values(5);           // [0 0 0 0 0]  
    fill(values.begin(), values.end(), 42);    // → [42 42 42 42 42]  
  
    array<int, 4> numbers;           // [?, ?, ?, ?]  
    fill_n(numbers.begin(), numbers.size(), -1); // → [-1 -1 -1 -1]  
  
    for (int v : values) cout << v << ' '; cout << "\n";    // 42 42 42 42 42  
  
    for (int n : numbers) cout << n << ' '; cout << "\n";    // -1 -1 -1 -1  
}
```

generate і generate_n

`generate` проходить по діапазону та для кожної позиції викликає передану функцію-генератор, записуючи її результат; `generate_n` робить це стільки разів, скільки вказано у `count`, і повертає кінець заповненого відрізка. Обидва алгоритми зручні, коли значення мають обчислюватись під час виконання програми (in runtime).

```
int main() {  
    vector<int> sequential(6);  
    int current = 1;  
    generate(sequential.begin(), sequential.end(), [&current]() { return current++; }); // → 1 2 3 4 5 6  
  
    vector<int> hundreds(4);  
    generate_n(hundreds.begin(), 3, []() { return 100; }); // заповнює перші 3 елементи  
  
    for (int v : sequential) cout << v << ' '; cout << '\n';           // 1 2 3 4 5 6  
  
    for (int v : hundreds)   cout << v << ' '; cout << '\n';           // 100 100 100 0  
}
```

generate random numbers

```
int main() {  
    vector<int> numbers(10); // 10 елементів  
  
    srand(time(nullptr)); // Ініціалізуємо генератор випадкових чисел  
  
    // Заповнюємо вектор випадковими числами від 1 до 100  
    generate(numbers.begin(), numbers.end(), []() {  
        return rand() % 100 + 1;  
    });  
  
    copy(numbers.begin(), numbers.end(), ostream_iterator<int>(cout, " "));  
  
    return 0;  
}
```

```
7 18 61 41 96 22 43 56 91 71
```


Перетворення



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Перетворення

Метод	Короткий опис
<code>transform(InputIt first, InputIt last, OutputIt d_first, UnaryOp op)</code>	Записує до вихідного діапазону результат одноаргументної функції <code>op</code> , застосованої до кожного елемента в <code>[first, last)</code> .
<code>transform(InputIt1 first1, InputIt1 last1, InputIt2 first2, OutputIt d_first, BinaryOp op)</code>	Викликає <code>op(x, y)</code> для парних елементів двох діапазонів і зберігає результати, створюючи нову послідовність.
<code>replace(ForwardIt first, ForwardIt last, const T& old_value, const T& new_value)</code>	Заміняє всі входження <code>old_value</code> у діапазоні на <code>new_value</code> .
<code>replace_if(ForwardIt first, ForwardIt last, UnaryPred pred, const T& new_value)</code>	Замінює елемент, якщо <code>pred(element)</code> повертає <code>true</code> .
<code>replace_copy(InputIt first, InputIt last, OutputIt d_first, const T& old_value, const T& new_value)</code>	Копіює діапазон, паралельно замінюючи <code>old_value</code> на <code>new_value</code> ; саме джерело не змінюється. Повертає ітератор за останній скопійований елемент
<code>replace_copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPred pred, const T& new_value)</code>	Створює копію діапазону, у якій елементи, що задовольняють <code>pred</code> , замінено на <code>new_value</code> . Повертає ітератор за останній скопійований елемент

transform

`transform` має дві форми. Однаргументна обробляє кожен елемент діапазону функцією `op` і записує результат у вихідний діапазон. Двоаргументна паралельно бере елементи з двох діапазонів, застосовує бінарну операцію `op(x, y)` і так само записує результат починаючи з `d_first`.

```
int main() {  
    vector<int> baseValues = {1, 2, 3};  
    vector<int> squared(baseValues.size());  
    transform(baseValues.begin(), baseValues.end(), squared.begin(), [](int value) { return value * value; });  
  
    vector<int> offsets = {10, 10, 10};  
    vector<int> sums(baseValues.size());  
    transform(baseValues.begin(), baseValues.end(), offsets.begin(), sums.begin(), [](int a, int b) { return a + b; });  
  
    for (int v : squared) cout << v << ' '; cout << '\n';           // 1 4 9  
  
    for (int v : sums) cout << v << ' '; cout << '\n';             // 11 12 13  
}
```

replace i replace_if

`replace` проходить по діапазону й підміняє кожне точне входження `old_value` на `new_value`. `replace_if` робить те саме, але застосовує заміну лише там, де предикат повертає `true`.

```
int main() {  
    vector<string> words = {"one", "two", "one"};  
    replace(words.begin(), words.end(), string("one"), string("once"));  
  
    vector<int> numbers = {1, 2, 3, 4, 5};  
    replace_if(numbers.begin(), numbers.end(), [](int value) { return value % 2 == 1; }, 0);  
  
    for (const string& w : words) cout << w << ' '; cout << "\n";    // once two once  
  
    for (int n : numbers) cout << n << ' '; cout << "\n";    // 0 2 0 4 0  
}
```

replace_copy і replace_copy_if

`replace_copy` створює копію діапазону й у процесі підміняє всі входження `old_value` на `new_value`, лишаючи вихідні дані без змін. `replace_copy_if` копіює діапазон, замінюючи тільки ті елементи, що задовольняють предикат.

```
int main() {  
    vector<int> source = {1, 2, 1, 3};  
    vector<int> replaced(source.size());  
    replace_copy(source.begin(), source.end(), replaced.begin(), 1, 99);  
  
    vector<int> replacedOdds;  
    replace_copy_if(source.begin(), source.end(), back_inserter(replacedOdds),  
        [](int value) { return value % 2 == 1; }, -1);  
  
    for (int v : replaced) cout << v << ' '; cout << '\n';           // 99 2 99 3  
  
    for (int v : replacedOdds) cout << v << ' '; cout << '\n';       // -1 2 -1 -1  
}
```

Видалення. remove-erase ідіома

Видалення

Метод	Короткий опис
<code>remove(ForwardIt first, ForwardIt last, const T& value)</code>	переміщує всі елементи, відмінні від <code>value</code> , на початок діапазону й повертає «новий кінець»; реального скорочення контейнера не відбувається.
<code>remove_if(ForwardIt first, ForwardIt last, UnaryPred pred)</code>	аналогічно <code>remove</code> , але відфільтровує елементи, для яких <code>pred(elem)</code> дорівнює <code>true</code> .
<code>remove_copy(InputIt first, InputIt last, OutputIt d_first, const T& value)</code>	копіює всі елементи, відмінні від <code>value</code> , у вихідний діапазон; вихідний контейнер залишається недоторканим.
<code>remove_copy_if(InputIt first, InputIt last, OutputIt d_first, UnaryPred pred)</code>	копіює лише ті елементи, що не задовольняють предикат, утворюючи «очищену» копію.
<code>unique(ForwardIt first, ForwardIt last)</code>	прибирає безпосередні дублікати, залишаючи по одному екземпляру кожної послідовної групи; повертає новий логічний кінець.
<code>unique(ForwardIt first, ForwardIt last, BinaryPred eq)</code>	те саме, але вважає елементи однаковими, якщо <code>eq(x, y)</code> повертає <code>true</code> .
<code>unique_copy(InputIt first, InputIt last, OutputIt d_first)</code>	створює копію без послідовних дублікатів; порядок зберігається.
<code>unique_copy(InputIt first, InputIt last, OutputIt d_first, BinaryPred eq)</code>	варіант із власним критерієм рівності.

Ідіома remove-erase

Алгоритми `remove` і `remove_if` лише пересувають «непотрібні» елементи в кінець і повертають ітератор нового логічного кінця, але фактичний розмір контейнера не змінюється. Щоб справді позбутися зайвих даних у послідовному контейнері (наприклад `vector` чи `string`), потрібно одразу викликати його метод `erase`, передавши отриману пару ітераторів. Така композиція називається *remove-erase* ідіомою.

```
int main() {  
    vector<int> numbers = {1, 2, 3, 2, 4, 2};  
    for (int value : numbers) cout << value << ' '; cout << '\n';    // 1 2 3 2 4 2  
  
    auto it = remove(numbers.begin(), numbers.end(), 2);  
    for (int value : numbers) cout << value << ' '; cout << '\n';    // 1 3 4 2 4 2  
  
    numbers.erase(it, numbers.end());  
    for (int value : numbers) cout << value << ' '; cout << '\n';    // 1 3 4  
}
```


remove та remove_if

`remove` переставляє всі елементи, відмінні від вказаного значення, на початок діапазону, а `remove_if` робить те саме, використовуючи предикат; обидва повертають новий логічний кінець, після якого розташовуються «зайві» елементи.

```
int main() {  
    vector<int> values = {1, 2, 3, 2, 4, 2};  
    auto newEnd = remove(values.begin(), values.end(), 2);  
    values.erase(newEnd, values.end());  
    vector<int> odds = {1, 2, 3, 4, 5};  
    odds.erase(remove_if(odds.begin(), odds.end(), [](int v) { return v % 2 == 0; }), odds.end());  
    for (int v : values) cout << v << ' '; cout << "\n";    // 1 3 4  
  
    for (int v : odds) cout << v << ' '; cout << "\n";    // 1 3 5  
}
```

STL реалізація remove_if

```
template<class ForwardIt, class UnaryPred>
ForwardIt remove_if(ForwardIt first, ForwardIt last, UnaryPred p)
{
    first = std::find_if(first, last, p);
    if (first != last)
        for (ForwardIt i = first; ++i != last;)
            if (!p(*i))
                *first++ = std::move(*i);
    return first;
}
```

remove_copy та remove_copy_if

`remove_copy` формує нову послідовність, копіюючи лише елементи, відмінні від указанного значення; `remove_copy_if` пропускає ті, що задовольняють предикат. Джерело не змінюється.

```
int main() {  
    vector<int> source = {1, 2, 1, 3};  
    vector<int> noOnes;  
    // копіюємо все, крім 1  
    remove_copy(source.begin(), source.end(), back_inserter(noOnes), 1);  
  
    vector<int> noOdds;  
    // копіюємо лише парні  
    remove_copy_if(source.begin(), source.end(), back_inserter(noOdds), [](int v) { return v % 2 == 1; });  
  
    for (int v : noOnes) cout << v << ' '; cout << "\n";    // 2 3  
    for (int v : noOdds) cout << v << ' '; cout << "\n";    // 2  
}
```

unique та unique_copy

`unique` стискає послідовність, прибираючи послідовні дублікати (залишає перший з кожної групи) й повертає новий логічний кінець; щоб скоротити контейнер, одразу викликають `erase`. `unique_copy` створює копію без суміжних дублікатів, лишаючи вихідні дані недоторканими.

```
int main() {  
    vector<int> data = {1, 1, 2, 2, 2, 3, 3};  
    data.erase(unique(data.begin(), data.end()), data.end());  
    vector<int> original = {4, 4, 5, 5, 6};  
    vector<int> condensed;  
    unique_copy(original.begin(), original.end(), back_inserter(condensed));  
    for (int v : data) cout << v << ' '; cout << "\n";    // 1 2 3  
    for (int v : condensed) cout << v << ' '; cout << "\n";    // 4 5 6  
}
```

Перестановка

Перестановка

Метод	Короткий опис
<code>reverse(BidirIt first, BidirIt last)</code>	Розвертає діапазон навпаки, працюючи in-place.
<code>reverse_copy(InputIt first, InputIt last, OutputIt d_first)</code>	Створює копію діапазону у зворотному порядку, не змінюючи оригінал.
<code>rotate(FwdIt first, FwdIt middle, FwdIt last)</code>	Циклічно зсуває елементи так, що middle стає новим first; блок [first, middle) переїжджає в кінець.
<code>rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutputIt d_first)</code>	Записує результат rotate у новий діапазон, залишаючи джерело без змін.
<code>shuffle(RandomIt first, RandomIt last, URBG& g)</code>	Перемішує елементи рівномірно-випадково, використовуючи генератор g; вимагає випадкових ітераторів.
<code>swap_ranges(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2)</code>	Попарно міняє місцями елементи двох діапазонів однакової довжини.
<code>iter_swap(ForwardIt a, ForwardIt b)</code>	Обмінює значення, на які вказують два ітератори.
<code>next_permutation(BidirIt first, BidirIt last)</code>	Переставляє діапазон у наступну лексикографічну перестановку; повертає false, якщо вже була остання.
<code>prev_permutation(BidirIt first, BidirIt last)</code>	Аналогічно генерує попередню перестановку; false, якщо була перша.

reverse і reverse_copy

`reverse` просто розгортає діапазон навспак без створення копій, тоді як `reverse_copy` робить те саме, але пише результат у новий контейнер, залишаючи вихідний недоторканим.

```
int main() {  
    vector<int> numbers = {1, 2, 3, 4};  
    reverse(numbers.begin(), numbers.end());  
  
    vector<int> source = {5, 6, 7};  
    vector<int> reversed(source.size());  
    reverse_copy(source.begin(), source.end(), reversed.begin());  
  
    for (int n : numbers) cout << n << ' '; cout << '\n';           // 4 3 2 1  
    for (int n : reversed) cout << n << ' '; cout << '\n';          // 7 6 5  
}
```

rotate і rotate_copy

`rotate` циклічно зсуває елементи так, що вказаний ітератор `middle` стає початком, а `rotate_copy` робить ту ж операцію у новий контейнер.

```
int main() {  
    vector<int> values = {1, 2, 3, 4, 5};  
    rotate(values.begin(), values.begin() + 2, values.end());  
  
    vector<int> src = {10, 20, 30, 40};  
    vector<int> rotated(src.size());  
    rotate_copy(src.begin(), src.begin() + 1, src.end(), rotated.begin());  
  
    for (int v : values) cout << v << ' '; cout << '\n';    // 3 4 5 1 2  
    for (int v : rotated) cout << v << ' '; cout << '\n';    // 20 30 40 10  
}
```


shuffle

`shuffle` рівномірно випадково перемішує елементи діапазону, використовуючи наданий генератор випадкових чисел.

```
int main() {  
    vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
  
    // Ініціалізація генератора випадкових чисел  
    unsigned seed = chrono::system_clock::now().time_since_epoch().count();  
    shuffle(v.begin(), v.end(), default_random_engine(seed));  
  
    // Виводимо перемішаний вектор  
    for (int n : v) {  
        cout << n << " ";  
    }  
  
    return 0;  
}
```

5 7 9 4 8 2 6 3 1

swap_ranges і iter_swap

`swap_ranges` міняє місцями елементи двох діапазонів однакової довжини, а `iter_swap` робить обмін лише для двох позицій.

```
int main() {  
    vector<int> left = {1, 2, 3};  
    vector<int> right = {4, 5, 6};  
    swap_ranges(left.begin(), left.end(), right.begin());  
  
    for (int n : left) cout << n << ' '; cout << '\n';    // 4 5 6  
    for (int n : right) cout << n << ' '; cout << '\n';    // 1 2 3  
  
    iter_swap(left.begin(), left.begin() + 2);  
  
    for (int n : left) cout << n << ' '; cout << '\n';    // 6 5 4  
    for (int n : right) cout << n << ' '; cout << '\n';    // 1 2 3  
}
```

next_permutation і prev_permutation

`next_permutation` перебудовує діапазон у наступну лексикографічну перестановку та повертає `false`, якщо поточна була останньою; `prev_permutation` робить крок у зворотному напрямку.

```
int main() {  
    vector<int> perm = {1, 2, 3};  
    next_permutation(perm.begin(), perm.end());  
    for (int v : perm) cout << v << ' '; cout << '\n';    // 1 3 2  
    prev_permutation(perm.begin(), perm.end());  
    for (int v : perm) cout << v << ' '; cout << '\n';    // 1 2 3  
  
    prev_permutation(perm.begin(), perm.end());  
    for (int v : perm) cout << v << ' '; cout << '\n';    // 3 2 1  
}
```

Розбиття / упорядкування



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Розбиття / упорядкування

Метод	Короткий опис
<code>partition(FwdIt first, FwdIt last, UnaryPred pred)</code>	Переставляє елементи так, що всі з <code>pred == true</code> опиняються перед рештою; порядок не зберігається.
<code>stable_partition(BidirIt first, BidirIt last, UnaryPred pred)</code>	Те саме, але зберігає відносний порядок обох груп; потребує двобічних ітераторів.
<code>partition_copy(InputIt first, InputIt last, OutputIt t_first, OutputIt f_first, UnaryPred pred)</code>	Розподіляє елементи на дві нові послідовності: правдиві пише з <code>t_first</code> , хибні — з <code>f_first</code> .
<code>sort(RandomIt first, RandomIt last)</code>	Сортує діапазон за operator< , складність $O(n \log(n))$; порядок еквівалентних елементів не гарантований.
<code>sort(RandomIt first, RandomIt last, Compare comp)</code>	Варіант із власним компаратором.
<code>stable_sort(RandomIt first, RandomIt last[, Compare comp])</code>	Сортує, гарантуючи, що еквівалентні елементи лишаються у вихідному порядку.
<code>partial_sort(RandomIt first, RandomIt middle, RandomIt last[, Compare comp])</code>	Ставить найменші (або згідно <code>comp</code>) елементи у <code>[first, middle)</code> , не гарантує порядок решти.
<code>partial_sort_copy(InputIt first, InputIt last, RandomIt d_first, RandomIt d_last[, Compare comp])</code>	Копіює у вихідний діапазон найменші <code>d_last - d_first</code> елементів, одразу сортує їх.
<code>nth_element(RandomIt first, RandomIt nth, RandomIt last[, Compare comp])</code>	Розміщує елемент, який був би <code>nth</code> у відсортованій послідовності, на позицію <code>nth</code> ; елементи перед ним не більші, за ним — не менші.
<code>is_sorted(ForwardIt first, ForwardIt last[, Compare comp])</code>	Перевіряє, чи вже відсортований діапазон.
<code>is_sorted_until(ForwardIt first, ForwardIt last[, Compare comp])</code>	Повертає перший елемент, який порушує впорядкованість.

partition і stable_partition

`partition` переставляє елементи так, щоб ті, для яких предикат повертає `true`, опинилися спереду, але при цьому порядок всередині групи не зберігається. `stable_partition` виконує ту ж операцію, але береже відносну послідовність елементів.

```
int main() {  
    auto isEven = [](int v) { return v % 2 == 0; };  
  
    vector<int> shuffled = {5, 2, 3, 6, 1, 4};  
    partition(shuffled.begin(), shuffled.end(), isEven);  
    for (int v : shuffled) cout << v << ' '; cout << '\n'; // 4 2 6 3 1 5 (порядок парних змішався)  
  
    vector<int> stable = {5, 2, 3, 6, 1, 4};  
    stable_partition(stable.begin(), stable.end(), isEven);  
    for (int v : stable) cout << v << ' '; cout << '\n';  
    // 2 6 4 5 3 1 (парні зберегли початковий порядок)  
}
```

partition_copy

`partition_copy` читає вихідний діапазон один раз і одразу викладає елементи, що проходять предикат, у перший вихідний контейнер, а решту — у другий.

```
int main() {  
    vector<int> raw = {1, 2, 3, 4, 5};  
    vector<int> evens, odds;  
    partition_copy(raw.begin(), raw.end(), back_inserter(evens), back_inserter(odds),  
        [](int v) { return v % 2 == 0; });  
  
    for (int v : evens) cout << v << ' '; cout << '\n'; // 2 4  
    for (int v : odds)  cout << v << ' '; cout << '\n'; // 1 3 5  
}
```

sort і stable_sort

`sort` впорядковує елементи за замовчанням через `operator<`, не піклуючись про порядок еквівалентів, тоді як `stable_sort` гарантує, що рівні елементи залишаться у тому ж відносному порядку.

```
int main() {  
    vector<int> numbers = {5, 1, 4, 1, 3};  
    sort(numbers.begin(), numbers.end());  
    for (int v : numbers) cout << v << ' '; cout << '\n'; // 1 1 3 4 5  
  
    struct Person { string name; int age; };  
    vector<Person> crowd = {{ "Ann", 30}, {"Bob", 25}, {"Amy", 25}};  
    stable_sort(crowd.begin(), crowd.end(),  
        [](const Person& a, const Person& b){ return a.age < b.age; });  
    for (const auto& p : crowd) cout << p.name << ' '; cout << '\n';  
    // Bob Amy Ann (Amy залишилася після Bob, бо так було спочатку)  
}
```


partial_sort і partial_sort_copy

`partial_sort` приводить перші `k` елементів до відсортованого стану, залишаючи решту невпорядкованою, тоді як `partial_sort_copy` одразу копіює найменші `k` елементів у потрібне місце.

```
int main() {  
    vector<int> data = {7, 2, 9, 4, 3};  
    partial_sort(data.begin(), data.begin() + 3, data.end()); // 2 3 4 x x  
    for (int v : data) cout << v << ' '; cout << '\n';  
    // 2 3 4 9 7 (лише перша трійка гарантовано відсортована)  
  
    vector<int> src = {8, 1, 6, 0, 5};  
    vector<int> top3(3);  
    partial_sort_copy(src.begin(), src.end(), top3.begin(), top3.end());  
    for (int v : top3) cout << v << ' '; cout << '\n'; // 0 1 5  
}
```

nth_element

`nth_element` швидко знаходить та ставить елемент, який матиме позицію `nth` у повністю відсортованій послідовності; усе зліва не більше, усе справа — не менше, але порядок усередині груп не визначений.

```
int main() {  
    vector<int> scores = {5, 1, 9, 3, 7};  
    nth_element(scores.begin(), scores.begin() + 2, scores.end());  
  
    cout << "Третій за величиною елемент: " << scores[2] << '\n'; // 5  
  
    for (int v : scores) cout << v << ' '; cout << '\n'; // 3 1 5 9 7 (ліва частина ≤5, права ≥5)  
}
```

is_sorted i is_sorted_until

`is_sorted` просто каже, чи вже впорядкований діапазон, тоді як `is_sorted_until` показує, де саме порушується порядок.

```
int main() {  
    vector<int> ok = {1, 2, 3};  
    vector<int> bad = {1, 4, 3, 5};  
    cout << boolalpha << is_sorted(ok.begin(), ok.end()) << '\n'; // true  
  
    auto firstWrong = is_sorted_until(bad.begin(), bad.end());  
    cout << *firstWrong << '\n'; // 3  
}
```

Дякую