Лекція 13. Алгоритми

1. *Концепція функтор та її реалізація. Об'єкт-функція.*
2. *Стандартні функтори, композиція функторів.*
3. *Каталог алгоритмів.*
4. *Приклади використання алгоритмів.*

На одній з попередніх лекцій ми розробили такі шаблони функцій для роботи з послідовностями (чи схожі до них):

```
// шаблон алгоритму відшукання елемента послідовності, що містить val
//   (вміє тільки шукати значення)
template <typename tIter>
tIter find_val(tIter start, tIter end, double val)
{
	for (tIter p = start; p! = end; ++p)
		if (*p == val) return p;
	return end;
}
// "просунутий" шаблон
//   (вміє знаходити елемент, для якого виконується деяка умова cond)
template <typename tIter, typename predicate>
tIter find_any(tIter start, tIter end, predicate cond)
{
	for (tIter p = start; p != end; ++p)
		if (cond(*p)) return p;
	return end;
}
```

Шаблон *find_any* узагальнює *find_val*, оскільки замість конкретної умови «елемент послідовності, на який вказує ітератор *p*, дорівнює значенню *val*» використовує абстракцію «*cond(*p)*». Що ж воно таке, оте *cond*? І що то за тип *predicate*?

Відповіді на ці запитання ми не знайдемо ні серед параметрів шаблона, ні в прототипі функції. Єдине, на що залишається сподіватися – блок функції. Видно, що *cond* використано з круглими дужками на місці логічного виразу. Схоже на виклик функції. Функції, що приймає один аргумент і повертає *bool*. В математичній логіці і програмуванні функції, чиїми значеннями є *true* або *false*, називають *предикатами*. Предикат може бути визначений на довільній предметній області, з довільною кількістю аргументів. Отже, *cond* – це *щось*, що схоже на унарний предикат; *щось*, до чого можна звертатися як до функції і сподіватися на результат логічного типу.

Що найбільше подібне на функцію? Функція! Шаблон *find_any* можна використати для відшукання першої по порядку п'ятірки в масиві дійсних чисел, або першого непарного елемента в масиві цілих:

```
bool equal_5(double x)    // перевіряє рівність аргумента числу 5
{	return x == 5.0;
}
bool odd(int x)           // перевіряє непарність аргумента
{	return x % 2;
}

double a[7] = { 1., 2., 3., 4., 5., 6., 7. };
int b[5] = { 4, 2, 0, 1, 3 };

double * p = find_any(a, a + 7, equal_5); // tIter == double*; predicate == bool (double)
int * q = find_any(b, b + 5, odd);        // tIter == int*; predicate == bool (int)
```

Трохи загальніший спосіб дає шаблон функції, якщо послідовність містить цілі числа, і конкретизацію шаблона можна задати ще перед виконанням:

```
template <int divisor>
bool divisibleBy(int x)   // перевіряє чи ділиться аргумент на число divisor
{      return x % divisor;
}
q = find_any(b, b + 5, divisibleBy<3>);  // перше кратне трьом
```

Ми знаємо й інші приклади сутностей, що ведуть себе, як функції: вказівник на функцію і лямбда-вираз. Попередні приклади можна було б переписати і так:

```
bool (*ptr)(double) = equal_5;  // оголошення функції потрібне!

double * p = find_any(a, a + 7, ptr);
int * q = find_any(b, b + 5, [](int x){ return x % 2; }); // лямбда замість odd
```

Вказівник на функцію гнучкіший за безпосереднє вказання імені функції, бо його значення може бути результатом попередніх обчислень, може допомагати відокремити незмінну частину коду від варіативної. Лямбда-вирази майже завжди кращі від оголошення дрібних «одноразових» функцій. Код стає виразнішим, не засмічується простір імен. Проте згадані приклади не вичерпують перелік сутностей, «схожих на функцію». Мова C++ дозволяє програмістові створити об'єкт-функцію за допомогою перевантаження оператора «круглі дужки».

Об'єкт-функція

У багатьох випадках зручніше використовувати не звичайну, а об'єкт-функцію. В тілі *equal_5* число 5 «прибито цвяшком», тому її не використаєш для відшукання інших чисел. Щоб знайти в масиві число 7 доведеться оголошувати іншу функцію, наприклад, *equal_7*, або використовувати лямбда-вираз. Це все можливо, якщо шукане значення відоме на етапі компіляції. А якщо значення отримують у результаті обчислень чи від користувача на етапі виконання – як тоді бути? На допомогу прийде клас з визначеним *operator()*. Екземпляр класу може зберігати додаткові дані, задані користувачем, і використовувати їх в тілі оператора. Розглянемо таке оголошення:

```
class pEqual              // об'єкт-функція
{private:
      double etalon;
public:
      explicit pEqual(double val = 0) :etalon(val) {}
      bool operator()(double x) const { return x == etalon; }
};

double * p = find_any(a, a + 7, pEqual(5.0)); // шукаємо 5
double * q = find_any(a, a + 7, pEqual(7.0)); // а тепер – 7
double x; cin >> x;
p = find_any(a, a + 7, pEqual(x));  // а тепер те, що захотів користувач
```

Виклик конструктора *pEqual* створює тимчасовий анонімний екземпляр, ініціалізує його поле *etalon* вказаним значенням і передає його алгоритмові *find_any*, щоб той використав екземпляр для виклику оператора () – для виконання порівнянь.

Оголошений клас *pEqual* є досить аскетичним: не містить ніяких зайвих методів, лише конструктор і оператор. Навіть складається враження, що це якийсь «неправильний» клас. Проте, з ним усе гаразд. Саме так виглядає типове оголошення об'єкт-функції. Для додаткових потреб можна було б ще додати метод зміни значення *etalon*, щоб була змога використовувати один і той самий екземпляр для відшукання різних значень.

Об'єкт-функція має декілька переваг, порівняно зі звичайною функцією:

1. *Це «функція зі станом»*. Наявність полів даних дозволяє налаштовувати стан об'єкта перед першим викликом і пам'ятати стан між викликами. Два екземпляри одного функціонального типу можуть перебувати в різних станах.

2. *Об'єкт-функція має свій тип*. Типи звичайних функцій збігаються, якщо вони мають однакові сигнатури. Тип-об'єкт функції – це її клас. Він не залежить від сигнатури оператора (). Кожна функціональна поведінка, задана за допомогою об'єкт-функції, має свій унікальний тип. Тому її можна передавати як значення параметра шаблону для налаштування поведінки алгоритмів і для конструювання поведінки контейнерів. Пригадайте, в оголошенні пріоритетної черги можна задавати параметр-компаратор – це функціональний об'єкт, що описує спосіб порівняння її елементів. Пріоритетні черги з різними компараторами мають *різний* тип.
3. *Об'єкт-функції швидші за звичайні*. Їхнє використання з шаблонами сприяє кращій оптимізації, що підвищує продуктивність програми.

Бібліотека STL містить низку стандартних шаблонів об'єкт-функцій і засобів для їхньої композиції. Про них поговоримо трохи згодом.

Функтори

Функтор, функціональний об'єкт – це концепція, що узагальнює можливість виклику: частина, елемент програми, до якого можна застосувати оператор «круглі дужки». Втілення такої концепції ми перелічили вище: звичайна функція (стандартна, чи оголошена користувачем), шаблон функції, лямбда-вираз, екземпляр класу, для якого визначено *operator()*. Функтори дають змогу уточнювати, налаштовувати поведінку алгоритмів і влаштування контейнерів.

Класифікація функторів

У бібліотеці функтори класифікують за кількістю параметрів і типом значення, що повертається:

- *Генератор (G)* – викликають без аргументу, отримують значення певного типу (наприклад, типу елементів контейнера).
- *Унарна функція (UF)* – один аргумент певного типу (наприклад, типу елементів контейнера), результат довільного типу (можливо і *void*).
- *Бінарна функція (BF)* – два аргументи будь-яких типів (можливо, різних), результат довільного типу (можливо, *void*).
- *Унарний предикат (UP)* – унарна функція, що повертає *bool*.
- *Бінарний предикат (BP)* – бінарна функція, що повертає *bool*.
- *Строга квазівпорядкованість (SO)* – бінарний предикат, що забезпечує розширене тлумачення поняття «рівність». Застосовують для тих значень, для яких оператор == ненадійний або недоступний.

Клас *pEqual* з нашого прикладу є унарним предикатом.

Шаблони стандартних функторів

Шаблони стандартних функторів параметризовано типом аргументів. Вони уміють повідомляти тип своїх аргументів і тип результату. Наприклад, дію додавання двох значень описано шаблоном

```
template <class T>
struct plus
{
	T operator()( const T& lhs, const T& rhs ) const
	{
		return lhs + rhs;
	}
};
```

У таблиці нижче подано лише імена шаблонів.

Ім'я	**Тип**	**Результат**
negate	Унарна функція	– arg
plus	Бінарна функція	arg1 + arg2
minus	Бінарна функція	arg1 – arg2
multiplies	Бінарна функція	arg1 * arg2
divides	Бінарна функція	arg1 / arg2
modulus	Бінарна функція	arg1 % arg2
equal_to	Бінарний предикат	arg1 == arg2
not_equal_to	Бінарний предикат	arg1 != arg2
greater	Бінарний предикат	arg1 > arg2
less	Бінарний предикат	arg1 < arg2
greater_equal	Бінарний предикат	arg1 >= arg2
less_equal	Бінарний предикат	arg1 <= arg2
logical_not	Унарний предикат	!arg
logical_and	Бінарний предикат	arg1 && arg2
logical_or	Бінарний предикат	arg1 || arg2
bit_and	Бінарна функція	arg1 & arg2
bit_or	Бінарна функція	arg1 | arg2
bit_xor	Бінарна функція	arg1 ^ arg2

З таких простих шаблонів-виразів можна будувати складніші за допомогою адаптерів

Вираз	**Результат**
bind(op, args...)	Функтор, що пов'язує список аргументів *args* з функтором *op*.
mem_fn(op)	Функтор, що може викликати метод (член класу) *op*.
not1(op)	Унарний предикат, результат якого протилежний до результату унарного предиката *op*.
not2(op)	Бінарний предикат, результат якого протилежний до результату бінарного предиката *op*.

Наприклад, аналогом функтора *pEqual(x)* буде конструкція *bind(equal_to<T>(), _1, x)*. Тут адаптер *bind* перетворює бінарну функцію *equal_to* на унарну. Перший параметр *equal_to* пов'язано з іменем *_1*, а другий – з *x*. Як це працюватиме? Величина *x* залишатиметься постійною – тою, яку задали при виклику *bind*. Ім'я *_1* позначає аргумент новоствореного функтора. Сюди потраплятимуть аргументи при звертанні до оператора (). Традиційний уже пошук п'ятірки в масиві можемо записати так:

```
double * p = find_any(a, a + 7, bind(equal_to<double>(), _1, 5.0)); // шукаємо 5
// елементи масиву a по черзі потраплятимуть в equal_to через параметр _1
```

Розглянемо складніший приклад. Як сконструювати бінарний предикат, що перевіряє значення виразу $-2 \le x \le 5$ для довільного цілого числа *x*? За допомогою *bind* і стандартних предикатів:

```
bind(logical_and<bool>(),
      bind(greater_equal<int>(), _1, -2), // _1 позначає x
      bind(less_equal<int>(), _1, 5))
```

Раніше для побудови композиції функторів використовували стандартні адаптери *bind1st()* та *bind2nd()*, що перетворюють бінарні функтори на унарні, підставляючи замість одного з їхніх параметрів постійне значення. У стандарті C++ 11 їх вважають застарілими.

Застарілими також стали адаптери *ptr_fun(function_name)*, *mem_fun(method_name)* і *mem_fun_ref(method_name)*, які вміють адаптувати звичайну функцію чи метод об'єкта до потреб алгоритмів STL.

Адаптери *not1* і *not2* – практично застарілі, бо без них завжди можна обійтися.

Узагальнені алгоритми

Ми вже знайомі з окремими алгоритмами. Наприклад, за допомогою алгоритму *copy* та різних ітераторів ми наповнювали контейнер значеннями, виводили його вміст у потік і, навпаки, завантажували з потоку, копіювали елементи з одного контейнера в інший. Ми мали б уже відчути, що програмування за допомогою алгоритмів – це щось більш загальне, більш абстрактне, ніж програмування за допомогою циклів. Замість того, щоб говорити «*переберемо елементи масиву по одному і виведемо їх на консоль*», ми тепер можемо сказати коротко «*скопіюємо вміст масиву на консоль*». Більше того, замість слова «масив» ми можемо вжити у тому ж реченні «список», «вектор», «множина» тощо. За допомогою алгоритмів ми можемо підняти свої програми на вищий рівень абстракції і суттєво скоротити час написання.

Усі алгоритми бібліотеки STL працюють з інтервалами, що задані ітераторами. Зазвичай використовують пару ітераторів: на початок інтервалу, на перший його елемент, і на наступний після останнього, за межі інтервалу. Якщо алгоритм працює з двома інтервалами, то перший задають парою ітераторів, а другий – лише ітератором на початок. Уважають, що другий інтервал містить стільки ж елементів, як і перший. За таких умов вказувати кінець другого інтервалу не потрібно. Алгоритми ***не перевіряють*** коректність меж інтервалів – це обов'язок користувача. Зокрема, інтервал для зберігання результатів роботи алгоритму мусить мати достатній розмір, щоб умістити всі. Алгоритми працюють у режимі *заміни*, а не вставки. Запис значення в інтервал за ітератором замінює те, яке було там перед записом. Для того, щоб алгоритм *вставляв* значення в інтервал, використовують спеціальні ітератори – ітератори вставляння (на початок, в середину, чи на кінець).

Кожен алгоритм бібліотеки є готовим ефективним рішенням певної задачі. На стільки ефективним, на скільки це можна зробити в загальному випадку. Іноді контейнери мають методи, що виконують таку ж роботу, що й алгоритми. В цьому випадку методи досягають вищої ефективності, оскільки реалізовані з урахуванням внутрішньої структури контейнера.

При іменуванні алгоритмів розробники дотримувались певних домовленостей. Щоб розуміти особливості алгоритму варто звертати увагу на суфікси та префікси, присутні в його імені. Наприклад, «однойменні» алгоритми *fill* і *fill_n* приймають різний набір аргументів. Перший з них – пару ітераторів, а другий – ітератор на початок інтервалу і ціле число, довжину інтервалу, задану кількістю елементів, які він містить.

Суфікс *_if* свідчить про те, що алгоритм приймає унарний або бінарний предикат. Такий предикат задає користувач, щоб описати потрібний йому критерій, який алгоритм застосує до елементів інтервалу (чи інтервалів). Суфікс *_copy* підказує, що результат роботи алгоритм перенесе в новий інтервал, а інтервал (чи інтервали) з вихідними даними залишаться без змін. Зрозуміло, що інтервалом для результатів можна призначити один з інтервалів вхідних даних – тоді відбудеться їхня заміна.

Префікс *stable_* додають іменам алгоритмів, що при впорядкуванні, переміщенні елементів інтервалу зберігають початковий взаємний порядок рівних за значенням. Пригадайте приклад з пріоритетною чергою і об'єктами класу *TodoItem*. Екземпляри, рівні в сенсі оператора порівняння, можуть бути насправді різними, а їхній взаємний порядок – важливим. Стабільні алгоритми зазвичай затратніші, ніж їхні звичайні аналоги.

Каталог алгоритмів STL

Нижче наведено спрощені прототипи більшості алгоритмів. У їхньому записі використано умовні позначення, введені для стислості. Позначення ітераторів з лекції 9: *II* – Input Iterator, *OI* – Output Iterator, *FI* – Forward Iterator, *BI* – Bidirectional Iterator, *RI* – Random Access Iterator. Позначення функторів уведено в попередньому параграфі.

Позначення не є стандартними! Це лише вигадка автора. І нікому не кажіть, що «римське два» – це «інпут ітератор».

Поділ алгоритмів на категорії умовний. Їх можна класифікувати за різними ознаками, проте класифікація не має жодного значення для успішного використання алгоритмів.

Алгоритми, що не модифікують дані

- Застосування дії до кожного елемента

```
UF for_each(II first, II last, UF f);
// застосовує f до кожного елемента [first,last), повертає f
```

Простий алгоритм, що може виявитися універсальним завдяки використанню різних функторів. Нехай *V* – вектор цілих чисел, тоді можна навести такі приклади використання:

```
struct Summ
{
	int s;
	Summ() :s(0) {}
	void operator()(int x) { s += x; }
};

	cout << "\n -- for_each is printing --\n\n";
	for_each(V.cbegin(), V.cend(), [](int x){ cout << '[' << x << ']'; });

	cout << "\n\n -- for_each is accumulating --\n\n";
	int summa = for_each(V.cbegin(), V.cend(), Summ()).s;
	cout << "summa = " << summa << '\n';
```

У другому прикладі *for_each* застосовує до кожного елемента вектора об'єкт-функцію, що здатна накопичувати суму. Оскільки алгоритм повертає як результат функтора, то й обчислене значення легко довідатися.

Алгоритм може перетворитися на такий, що модифікує дані, якщо йому передати відповідний функтор, але про це трохи згодом.

Важливо: предикати, використані з наступними алгоритмами, не повинні змінювати свого стану! Якщо предикатом є об'єкт-функція, то її *operator()* мав би бути константним.

- Підрахунок

```
IntegralValue count(II first, II last, const EqualityComparable& val);
// кількість елементів з [first,last), що == val
IntegralValue count_if(II first, II last, UP pred);
// кількість елементів з [first,last), для яких pred повертає true
```

- Пошук

```
II find(II first, II last, const EqualityComparable& val);
// повертає ітератор на елемент ==val або last
II find_if(II first, II last, UP pred);
// повертає ітератор на перший елемент, що задовольняє pred; або last
II find_if_not(II first, II last, UP pred);
// повертає ітератор на перший елемент, що НЕ задовольняє pred; або last
FI adjacent_find(FI first, FI last);
FI adjacent_find(FI first, FI last, BP pred);
// шукає пару однакових сусідів
FI1 find_first_of(FI1 first, FI1 last, FI2 fst, FI2 lst);
FI1 find_first_of(FI1 first, FI1 last, FI2 fst, FI2 lst, BP pred);
// шукають в [first,last) перший елемент з [first,last)∩[fst,lst)
FI1 search(FI1 first, FI1 last, FI2 fst, FI2 lst);
FI1 search (FI1 first, FI1 last, FI2 fst, FI2 lst, BP pred);
// шукають в [first,last) перше входження [fst,lst)
FI1 find_end(FI1 first, FI1 last, FI2 fst, FI2 lst);
FI1 find_end (FI1 first, FI1 last, FI2 fst, FI2 lst, BP pred);
// шукають в [first,last) останнє входження [fst,lst)
FI min_element(FI first, FI last);
FI min_element(FI first, FI last, BP pred);
```

```
FI max_element(FI first, FI last);
FI max_element(FI first, FI last, BP pred);
// шукають перший найменший (найбільший) елемент
pair<FI,FI> minmax_element(FI first, FI last);
pair<FI,FI> minmax_element(FI first, FI last, BP pred);
// шукають перший найменший і ОСТАННІЙ найбільший елементи
```

- Порівняння інтервалів

```
bool equal(II1 first, II1 last, II2 first2);
bool equal(II1 first, II1 last, II2 first2, BP pred);
// перевіряє два інтервали на рівність елементів
bool lexicographical_compare(II1 first, II1 last, II2 first2, II2 last2);
bool lexicographical_compare(II1 first, II1 last, II2 first2, II2 last2, BP pred);
// перевіряє, чи менший перший інтервал за другий
pair<II1,II2> mismatch(II1 first, II1 last, II2 first2);
pair<II1,II2> mismatch(II1 first, II1 last, II2 first2, BP pred);
// повертає першу пару елементів діапазонів, що відрізняються
bool is_permutation(FI1 first, FI1 last, FI2 first2);
bool is_permutation(FI1 first, FI1 last, FI2 first2, BP pred);
// перевіряє два інтервали на рівність елементів з точністю до перестановки
// «невпорядкована рівність»
bool is_sorted(FI first, FI last);
bool is_sorted(FI first, FI last, BP pred);
// перевіряє, чи впорядковано інтервал
FI is_sorted_until(FI first, FI last);
FI is_sorted_until(FI first, FI last, BP pred);
// знаходить першу позицію, в якій порушується впорядкованість
bool all_of(II first, II last, UP pred);
bool any_of(II first, II last, UP pred);
bool none_of(II first, II last, UP pred);
// перевіряють, чи повертає предикат true для кожного / хоча б для одного / для жодного
// елемента діапазону
```

Модифікуючі алгоритми

- Застосування дії до кожного елемента

```
UF for_each(II first, II last, UF f);
// застосовує f до кожного елемента [first,last), повертає f
```

Нехай *V* – вектор цілих чисел, тоді подвоїти значення його елементів можна так:

```
for_each(V.begin(), V.end(), [](int &x){ x *= 2; });
```

Фокус в тому, що аргументом функтора є *посилання* на елемент діапазона. Такий функтор може змінювати елементи контейнера.

```
OI transform(II first, II last, OI dest, UF f);
// застосовує f до кожного елемента [first,last), результати заносить в [dest,_)
OI transform(II1 first1, II1 last1, II2 first2, OI dest, BF f);
// застосовує f до кожної пари елементів з [first1,last1) і [first2,_),
// результати заносить в [dest,_)
```

- Заповнення інтервалів та генерування значень

```
void fill(FI first, FI last, const T& val);
// заповнює [first,last) значеннями val
void fill_n(OI first, Size n, const T& val);
// присвоює val n елементам, починаючи з first
void generate(FI first, FI last, G gen);
// заповнює [first,last) значеннями, згенерованими gen
void generate_n(OI first, Size n, G gen);
// присвоює згенероване gen значення n елементам, починаючи з first
```

- Копіювання

```
// всі алгоритми повертають ітератор «після останнього скопійованого»

OI copy(II first, II last, OI dest);
// копіює [first,last) в dest; dest∉[first,last); dest має досить місця
// щоб вставляти в порожній контейнер, чи дописувати в кінець, використовують
// ітератори вставляння для контейнера
OI copy_if(II first, II last, OI dest, UP pred);
// Умовне копіювання
OI copy_n(II first, Size n, OI dest);
// Діапазон копіювання задано початком і кількістю
BI2 copy_backward(BI1 first, BI1 last, BI2 destEnd);
// копіює в оберненому порядку, приймач заповнює з кінця
OI move(II first, II last, OI dest);
BI2 move_backward(BI1 first, BI1 last, BI2 dest);
// переміщає [first,last) в dest;
FI2 swap_ranges(FI1 first, FI1 last, FI2 dest);
// попарно міняє місцями елементи двох інтервалів однакового розміру
```

- Заміна

```
void replace(FI first, FI last, const T& old_value, const T& new_value);
void replace_if(FI first, FI last, UP pred, const T& new_value);
OI replace_copy(FI first, FI last, OI dest, const T& old_value, const T& new_value);
OI replace_copy_if(FI first, FI last, OI dest, UP pred, const T& new_value);
// заміняють на new_value ті значення з інтервалу [first,last), які ==old_value, або
// pred повернув true; результат – на місці, або в dest
```

- Перестановки

```
void reverse(BI first, BI last);
// обертає порядок елементів [first,last)
OI reverse_copy(BI first, BI last, OI dest);
// виготовляє обернену копію [first,last)
void rotate(FI first, FI middle, FI last);
// переміщує [first,middle) в кінець інтервалу, а (middle,last) – на початок
OI rotate_copy(FI first, FI middle, FI last, OI dest);
// робить те саме, що rotate, але в новому місці dest
bool next_permutation(BI first, BI last);
bool next_permutation(BI first, BI last, SO binary_pred);
bool prev_permutation(BI first, BI last);
bool prev_permutation(BI first, BI last, SO binary_pred);
// повертають true, якщо вдалось виконати наступну(попередню) перестановку елементів
// інтервалу (порядок перестановок – лексикографічний, за зростанням); порівняння -
// оператором < або бінарним предикатом
void random_shuffle(RI first, RI last);
void random_shuffle(RI first, RI last, G random_number);
// випадкова перестановка інтервалу
BI partition(BI first, BI last, UP pred);
BI stable_partition(BI first, BI last, UP pred);
// переміщають на початок інтервалу елементи, що задовольняють pred
// другий алгоритм зберігає початковий порядок елементів
pair<OI1,OI2> partition_copy(II first, II last, OI1 firtTrue, OI2 firstFalse, UP pred);
// поділ на два інтервали
```

- Вилучення елементів

```
/*Алгоритми вилучення переміщають «вилучені» елементи в кінець інтервала. Щоб вилучити їх
остаточно, використайте метод resize() контейнера. Залишені елементи зберігають початковий
взаємний порядок.*/

FI remove(FI first, FI last, const T& val);
FI remove_if(FI first, FI last, UP pred);
OI remove_copy(II first, II last, OI dest, const T& val);
OI remove_copy_if(II first, II last, OI dest, UP pred);
```

```
// вилучають елементи, що рівні заданому значенню, або задовольняють предикат
// варіант з _copy поміщає результат в нове місце
FI unique(FI first, FI last);
FI unique(FI first, FI last, BP pred);
OI unique_copy(II first, II last, OI dest);
OI unique_copy(II first, II last, OI dest, BP pred);
// знаходять в інтервалі пари однакових сусідів і вилучають дублікати
// щоб всі значення в контейнері зустрічалися рівно один раз, спочатку
// впорядкуйте його алгоритмом sort
```

- Впорядкування та споріднені дії

```
void sort(RI first, RI last);
void sort(RI first, RI last, SO binary_pred);
// впорядковує інтервал за зростанням, або за предикатом; взаємний порядок
// рівних елементів може змінитися
void stable_sort(RI first, RI last);
void stable_sort(RI first, RI last, SO binary_pred);
// впорядковує інтервал за зростанням, або за предикатом; взаємний порядок
// рівних елементів зберігається
void partial_sort(RI firs, RI middle, RI last);
void partial_sort(RI firs, RI middle, RI last, SO binary_pred);
// впорядковує елементи [first,last), що можуть за кількістю увійти в [first,middle),
// решту поміщає в [middle,last)
RI partial_sort_copy(II firs, II last, RI r_first, RI r_last);
RI partial_sort_copy(II firs, II last, RI r_first, RI r_last, SO binary_pred);
// впорядковує елементи [first,last), що можуть за кількістю увійти в
// [r_first,r_middle), результат впорядкування в [r_first,r_middle)
void nth_element(RI firs, RI nth, RI last);
void nth_element(RI firs, RI middle, RI nth, SO binary_pred);
// nth задає точку розбиття контейнера: ліворуч менші, праворуч – всі інші

bool binary_search(FI firs, FI nth, RI last, const T& val);
bool binary_search(FI firs, FI nth, RI last, const T& val, SO binary_pred);
// перевіряє, чи є задане значення у впорядкованому інтервалі
FI lower_bound(FI firs, FI nth, RI last, const T& val);
FI lower_bound(FI firs, FI nth, RI last, const T& val, SO binary_pred);
// повертає ітератор на перше входження val у впорядкований інтервал або last
FI upper_bound(FI firs, FI nth, RI last, const T& val);
FI upper_bound(FI firs, FI nth, RI last, const T& val, SO binary_pred);
// повертає ітератор після останнього входження val у впорядкований інтервал або last

Якщо алгоритмам пошуку передавати forward iterator, матимемо лінійну складність, а якщо
random access iterator – логарифмічну.

OI merge(II1 first1, II1 last1, II2 first2, II2 last2, OI dest);
OI merge(II1 first1, II1 last1, II2 first2, II2 last2, OI dest, SO binary_pred);
// об'єднує впорядковані [first1,last1) [first2,last2) в один впорядкований [dest,_)
// для порівняння обов'язково застосовувати той же критерій, щой для впорядкування
void inplace_merge(BI first, BI middle, BI last);
void inplace_merge(BI first, BI middle, BI last, SO binary_pred);
// об'єднує дві суміжні впорядковані частини одного контейнера

З упорядкованими інтервалами можна виконувати математичні операції з теорії множин
bool includes(II1 first1, II1 last1, II2 first2, II2 last2);
bool includes(II1 first1, II1 last1, II2 first2, II2 last2, SO binary_pred);
// повертає [first2,last2)⊂[first1,last1)
OI set_union(II1 first1, II1 last1, II2 first2, II2 last2, OI dest);
OI set_union(II1 first1, II1 last1, II2 first2, II2 last2, OI dest, SO binary_pred);
// обчислює [dest,_)= [first1,last1)∪[first2,last2)
OI set_intersection(II1 first1, II1 last1, II2 first2, II2 last2, OI dest);
OI set_intersection(II1 first1,II1 last1,II2 first2,II2 last2,OI dest,SO binary_pred);
// обчислює [dest,_)= [first1,last1)∩[first2,last2)
OI set_difference(II1 first1, II1 last1, II2 first2, II2 last2, OI dest);
```

```
OI set_difference(II1 first1,II1 last1,II2 first2,II2 last2,OI dest,SO binary_pred);
OI set_symmetric_difference(II1 first1, II1 last1, II2 first2, II2 last2, OI dest);
OI set_symmetric_difference(II1 first1, II1 last1, II2 first2, II2 last2, OI dest,
       SO binary_pred);
// обчислюють різницю та симетричну різницю двох інтервалів
```

- Числові алгоритми з <numeric>

```
T accumulate(II first, II last, T init);
T accumulate(II first, II last, T init, BF op);
T inner_product(II1 first1, II1 last1, II2 first2, II2 last2, T init);
T inner_product(II1 first1, II1 last1, II2 first2, II2 last2, T init,BF1 op1,BF2 op2);
OI adjacent_difference(II first, II last, OI result);
OI adjacent_difference(II first, II last, OI result, BF op);
```

Більше алгоритмів і прикладів до них у [Стандартная библиотека C++. Справочное руководство] на Диску.