

Міністерство освіти і науки України  
Національний університет «Львівська політехніка»  
кафедра інформаційних систем та мереж

Григорович Віктор

# **Об'єктно-орієнтоване програмування STL – стандартна бібліотека шаблонів**

Навчальний посібник

2021

Григорович Віктор Геннадійович

**Об'єктно-орієнтоване програмування. STL** – стандартна бібліотека шаблонів.

Навчальний посібник.

Дисципліна «Об'єктно-орієнтоване програмування» вивчається після курсу «Алгоритмізація та програмування», цією дисципліною продовжується цикл предметів, що стосуються програмування та розробки програмного забезпечення.

В посібнику містяться теоретичні відомості, приклади, методичні вказівки з їх розв'язування, варіанти лабораторних завдань та питання і завдання з контролю знань з теми «STL – стандартна бібліотека шаблонів».

Розглядаються наступні роботи лабораторного практикуму:

Лабораторна робота № 7.1.

Контейнери-масиви

Лабораторна робота № 7.2.

Контейнери-списки

Лабораторна робота № 7.3.

Контейнери-множини

Лабораторна робота № 7.4.

Контейнери-відображення

Відповідальний за випуск – Григорович В.Г.

## Стислий зміст

Вступ.....	15
Тема 7. STL – стандартна бібліотека шаблонів.....	16
Стисло та головне про STL – стандартну бібліотеку шаблонів .....	16
Контейнери .....	16
Ітератори .....	18
Алгоритми.....	22
Теоретичні відомості.....	23
Нові можливості C++ .....	23
STL – стандартна бібліотека шаблонів .....	32
Лабораторний практикум .....	156
Оформлення звіту про виконання лабораторних робіт .....	156
Лабораторна робота № 7.1. Контейнери-масиви .....	158
Лабораторна робота № 7.2. Контейнери-списки.....	166
Лабораторна робота № 7.3. Контейнери-множини.....	173
Лабораторна робота № 7.4. Контейнери-відображення .....	184
Питання та завдання для контролю знань .....	200
Контейнери .....	200
Ітератори .....	200
Алгоритми.....	200
Предметний покажчик .....	202
Література .....	204

## Зміст

Вступ.....	15
Тема 7. STL – стандартна бібліотека шаблонів.....	16
Стисло та головне про STL – стандартну бібліотеку шаблонів .....	16
Контейнери .....	16
Вимоги до елементів контейнера.....	17
Винятки .....	18
Ітератори .....	18
Категорії ітераторів.....	18
Адаптери ітераторів .....	19
Допоміжні функції для ітераторів .....	21
Алгоритми.....	22
Теоретичні відомості.....	23
Нові можливості C++ .....	23
Автоматичне виведення типу.....	23
Синтаксис.....	23
Пояснення .....	23
Приклади.....	24
Цикл <code>for</code> на основі діапазону .....	25
Лямбда-вирази .....	27
Секція фіксації.....	28
Список параметрів .....	28
Тип результату.....	28
Тіло лямбда-виразу .....	29
Приклади.....	30
STL – стандартна бібліотека шаблонів .....	32
Контейнери .....	32
Вимоги до елементів контейнера.....	33
Винятки .....	34
Послідовні контейнери .....	34
<code>array</code> .....	36
<code>vector</code> .....	37
<code>deque</code> .....	38
<code>forward_list</code> .....	39
<code>list</code> .....	39

Асоціативні контейнери .....	41
set .....	41
erase.....	41
insert.....	42
map .....	43
Невпорядковані асоціативні контейнери .....	44
Адаптери контейнерів.....	45
stack.....	45
queue.....	45
priority_queue .....	45
Втрата ітератора .....	47
Таблиці версій.....	49
Ітератори .....	55
Визначення ітератора.....	55
Категорії ітераторів.....	55
Адаптери ітераторів .....	59
Допоміжні функції для ітераторів .....	63
Концепти ітераторів C++20.....	63
Функції .....	64
Операції з ітераторами.....	64
advance .....	64
distance .....	65
begin / end.....	65
rbegin / rend .....	66
prev.....	67
next.....	67
Генератори ітераторів .....	68
back_inserter .....	68
front_inserter .....	69
inserter .....	69
make_move_iterator .....	70
Класи.....	71
iterator .....	71
iterator_traits.....	72
Визначені в STL ітератори .....	73

reverse_iterator.....	73
move_iterator .....	74
back_insert_iterator.....	75
front_insert_iterator.....	75
insert_iterator.....	76
istream_iterator.....	77
ostream_iterator.....	77
istreambuf_iterator .....	78
ostreambuf_iterator .....	79
Алгоритми.....	80
Операції, які не модифікують послідовність.....	80
accumulate.....	81
all_of.....	81
any_of.....	81
none_of .....	82
for_each .....	82
find.....	83
find_if .....	84
find_if_not.....	84
find_end .....	85
find_first_of .....	86
adjacent_find .....	87
count.....	88
count_if .....	89
mismatch .....	89
equal.....	90
is_permutation.....	91
search.....	92
search_n .....	93
Операції, які модифікують послідовність.....	95
copy .....	95
copy_n.....	96
copy_if .....	97
copy_backward .....	97

move.....	98
move_backward.....	99
swap.....	100
swap_ranges.....	100
iter_swap.....	101
transform.....	102
replace.....	103
replace_if.....	103
replace_copy.....	104
replace_copy_if.....	105
fill.....	105
fill_n.....	106
generate.....	106
generate_n.....	107
remove.....	108
remove_if.....	109
remove_copy.....	109
remove_copy_if.....	110
unique.....	111
unique_copy.....	112
reverse.....	113
reverse_copy.....	113
rotate.....	114
rotate_copy.....	114
random_shuffle.....	115
shuffle.....	116
Розбиття.....	118
is_partitioned.....	118
partition.....	119
stable_partition.....	120
partition_copy.....	120
partition_point.....	121
Сортування.....	123
sort.....	123

stable_sort.....	124
partial_sort .....	125
partial_sort_copy .....	126
is_sorted.....	127
is_sorted_until.....	127
nth_element.....	128
Бінарний пошук.....	130
lower_bound.....	130
upper_bound.....	130
equal_range.....	131
binary_search .....	132
Злиття .....	134
merge.....	134
inplace_merge .....	135
includes .....	136
set_union.....	136
set_intersection.....	137
set_difference.....	138
set_symmetric_difference.....	140
Максимальна купа.....	142
push_heap .....	142
pop_heap .....	143
make_heap.....	143
sort_heap.....	144
is_heap .....	145
is_heap_until .....	146
Min / max .....	147
min .....	147
max .....	147
minmax.....	148
min_element.....	148
max_element.....	149
minmax_element.....	150
Лексикографічні операції .....	152



lexicographical_compare .....	152
next_permutation.....	153
prev_permutation.....	154
Лабораторний практикум .....	156
Оформлення звіту про виконання лабораторних робіт .....	156
Вимоги до оформлення звіту про виконання лабораторних робіт №№ 7.1–7.4 .....	156
Зразок оформлення звіту про виконання лабораторних робіт №№ 7.1–7.4 .....	157
Лабораторна робота № 7.1. Контейнери-масиви .....	158
Мета роботи .....	158
Питання, які необхідно вивчити та пояснити на захисті.....	158
Приклад виконання завдання .....	158
Спосіб 1. ....	158
Спосіб 2. ....	159
Варіанти завдань .....	160
Варіант 1.....	160
Варіант 2.....	160
Варіант 3.....	160
Варіант 4.....	160
Варіант 5.....	160
Варіант 6.....	161
Варіант 7.....	161
Варіант 8.....	161
Варіант 9.....	161
Варіант 10.....	161
Варіант 11.....	161
Варіант 12.....	161
Варіант 13.....	161
Варіант 14.....	161
Варіант 15.....	162
Варіант 16.....	162
Варіант 17.....	162
Варіант 18.....	162
Варіант 19.....	162
Варіант 20.....	162
Варіант 21.....	162

Варіант 22.....	162
Варіант 23.....	163
Варіант 24.....	163
Варіант 25.....	163
Варіант 26.....	163
Варіант 27.....	163
Варіант 28.....	163
Варіант 29.....	163
Варіант 30.....	163
Варіант 31.....	163
Варіант 32.....	164
Варіант 33.....	164
Варіант 34.....	164
Варіант 35.....	164
Варіант 36.....	164
Варіант 37.....	164
Варіант 38.....	164
Варіант 39.....	164
Варіант 40.....	164
Лабораторна робота № 7.2. Контейнери-списки.....	166
Мета роботи .....	166
Питання, які необхідно вивчити та пояснити на захисті.....	166
Приклад виконання завдання .....	166
Спосіб 1. ....	166
Спосіб 2. ....	167
Варіанти завдань .....	168
Варіант 1.....	168
Варіант 2.....	168
Варіант 3.....	168
Варіант 4.....	168
Варіант 5.....	168
Варіант 6.....	168
Варіант 7.....	169
Варіант 8.....	169
Варіант 9.....	169

Варіант 10.....	169
Варіант 11.....	169
Варіант 12.....	169
Варіант 13.....	169
Варіант 14.....	169
Варіант 15.*.....	169
Варіант 16.*.....	170
Варіант 17.*.....	170
Варіант 18.....	170
Варіант 19.....	170
Варіант 20.....	170
Варіант 21.**.....	170
Варіант 22.**.....	170
Варіант 23.**.....	170
Варіант 24.**.....	170
Варіант 25.*.....	170
Варіант 26.....	171
Варіант 27.....	171
Варіант 28.....	171
Варіант 29.....	171
Варіант 30.....	171
Варіант 31.....	171
Варіант 32.....	171
Варіант 33.....	171
Варіант 34.....	171
Варіант 35.....	171
Варіант 36.....	172
Варіант 37.....	172
Варіант 38.....	172
Варіант 39.....	172
Варіант 40.*.....	172
Лабораторна робота № 7.3. Контейнери-множини.....	173
Мета роботи .....	173
Питання, які необхідно вивчити та пояснити на захисті.....	173
Приклад виконання завдання .....	173

Спосіб 1.....	173
Спосіб 2.....	175
Варіанти завдань .....	177
Варіант 1.....	177
Варіант 2.....	177
Варіант 3.....	177
Варіант 4.....	177
Варіант 5.....	177
Варіант 6.....	178
Варіант 7.....	178
Варіант 8.....	178
Варіант 9.....	178
Варіант 10.....	178
Варіант 11.....	178
Варіант 12.....	179
Варіант 13.....	179
Варіант 14.....	179
Варіант 15.....	179
Варіант 16.....	179
Варіант 17.....	179
Варіант 18.....	180
Варіант 19.....	180
Варіант 20.....	180
Варіант 21.....	180
Варіант 22.....	180
Варіант 23.....	180
Варіант 24.....	181
Варіант 25.....	181
Варіант 26.....	181
Варіант 27.....	181
Варіант 28.....	181
Варіант 29.....	181
Варіант 30.....	182
Варіант 31.....	182
Варіант 32.....	182

Варіант 33.....	182
Варіант 34.....	182
Варіант 35.....	182
Варіант 36.....	183
Варіант 37.....	183
Варіант 38.....	183
Варіант 39.....	183
Варіант 40.....	183
Лабораторна робота № 7.4. Контейнери-відображення .....	184
Мета роботи .....	184
Питання, які необхідно вивчити та пояснити на захисті.....	184
Приклад виконання завдання .....	184
Варіант 0.....	184
Розв'язок .....	184
Варіанти завдань .....	187
Варіант 1.....	187
Варіант 2.....	188
Варіант 3.....	188
Варіант 4.....	188
Варіант 5.....	189
Варіант 6.....	189
Варіант 7.....	189
Варіант 8.....	190
Варіант 9.....	190
Варіант 10.....	190
Варіант 11.....	190
Варіант 12.....	191
Варіант 13.....	191
Варіант 14.....	191
Варіант 15.....	192
Варіант 16.....	192
Варіант 17.....	192
Варіант 18.....	192
Варіант 19.....	193
Варіант 20.....	193

Варіант 21.....	193
Варіант 22.....	194
Варіант 23.....	194
Варіант 24.....	194
Варіант 25.....	195
Варіант 26.....	195
Варіант 27.....	195
Варіант 28.....	196
Варіант 29.....	196
Варіант 30.....	196
Варіант 31.....	197
Варіант 32.....	197
Варіант 33.....	197
Варіант 34.....	197
Варіант 35.....	198
Варіант 36.....	198
Варіант 37.....	198
Варіант 38.....	199
Варіант 39.....	199
Варіант 40.....	199
Питання та завдання для контролю знань .....	200
Контейнери .....	200
Ітератори .....	200
Алгоритми.....	200
Предметний показчик .....	202
Література .....	204

# Вступ

Дисципліна «Об'єктно-орієнтоване програмування» вивчається після курсу «Алгоритмізація та програмування», цією дисципліною продовжується цикл предметів, що стосуються програмування та розробки програмного забезпечення.

В посібнику містяться теоретичні відомості, приклади, методичні вказівки з їх розв'язування, варіанти лабораторних завдань та питання і завдання з контролю знань з теми «STL – стандартна бібліотека шаблонів».

# Тема 7. STL – стандартна бібліотека шаблонів

## Стисло та головне про STL – стандартну бібліотеку шаблонів

В цій темі розглядаються основи стандартної бібліотеки шаблонів (STL – standard template library). Розглядаються послідовні та асоціативні контейнери, ітератори, стандартні алгоритми та функтори.

Основні складові стандартної бібліотеки шаблонів – це *контейнери*, *ітератори* та *алгоритми*. Основна концепція STL – відокремлення даних від операцій. Дані зберігаються в контейнерах, операції визначаються адаптованими алгоритмами. Ітератори поєднують ці два компоненти, – завдяки ним будь-який алгоритм може працювати майже з будь-яким контейнером. Універсальність алгоритмів стає ще більшою при використанні *функторів*, які реалізовані в STL.

Важливою особливістю STL є те, що всі компоненти працюють з довільними типами, – бо вони оформлені у вигляді шаблонів.

### Контейнери

*Контейнер STL* – це набір однотипних елементів. Всі контейнери поділяються на наступні види – послідовні, асоціативні та неупорядковані асоціативні контейнери:

- *Послідовні контейнери* – це неупорядковані колекції однотипних елементів. Кожний елемент розташований в контейнері у певній позиції, яка залежить лише від часу та місця вставки, і не залежить від значення елемента. До послідовних контейнерів відносяться вектор (*vector*), список (*list*) і двостороння черга (*deque*).
- *Асоціативні контейнери* – це впорядковані за умовчанням колекції однотипних елементів, в яких позиція елемента залежить від його значення. До асоціативних контейнерів належать множина (*set*), мультимножина (*multiset* – множина з елементами, які повторюються), відображення (*map*) та мультівідображення (*multimap* – відображення з елементами, які повторюються).
- *Неупорядковані асоціативні контейнери* – добавлені в нових версіях C++. Це *unordered\_set*, *unordered\_map*, *unordered\_multiset*, *unordered\_multimap*.

Крім вище зазначених основних контейнерів, STL ще містить набір спеціальних контейнерів-адаптерів: стек (*stack*), черга (*queue*) та черга з пріоритетами (*priority\_queue*).



В бібліотеці STL реалізовані інші структури, які формально не є стандартними контейнерами елементів: булевий вектор (`vector<bool>`) та бітові поля (`bitset`).

**Зауваження:** STL в різних системах програмування може містити інші, нестандартні, контейнери.

Всі контейнери – динамічні: один із параметрів шаблону визначає розподіл пам'яті.

Для використання контейнерів та адаптерів слід підключити файли заголовків:

```
#include <vector>    // контейнер-вектор та vector<bool>
#include <deque>     // контейнер-двостороння черга
#include <list>      // контейнер-список
#include <stack>     // стек
#include <queue>     // черга та черга з пріоритетами
#include <set>       // множина та мультимножина
#include <map>       // відображення та мультивідображення
#include <bitset>    // послідовність бітів
```

## **Вимоги до елементів контейнера**

Елементи STL-контейнера можуть бути довільного типу `T`, але цей тип має задовольняти вимогам:

1. Всі контейнери створюють внутрішні копії своїх елементів та повертають ці копії. Тому `T` має мати конструктор копіювання. Копія має дорівнювати оригіналу, і поведінка копії не має відрізнятися від поведінки оригіналу.
2. Контейнери використовують операцію присвоєння для заміни старих елементів новими. Це означає, що тип (клас) `T` має підтримувати операцію присвоєння.
3. Контейнери знищують свої внутрішні копії при видаленні елементів із контейнера. Тому деструктор `T` не може бути закритим (`private`).

Ці три операції (копіювання, присвоєння, видалення) автоматично генеруються для будь-якого класу, якщо відсутні їх явні визначення. Тому будь-який клас за умовчанням задовольняє цим вимогам, якщо для нього не перевизначати цих операцій. Цим вимогам також задовольняють всі вбудовані типи.

Проте цим вимогам не задовольняють елементи бітових полів та булевих векторів – тому вони не вважаються стандартними контейнерами.

Стандартні вбудовані вказівники теж не повністю задовольняють зазначеним вимогам, тому використовувати вказівники в якості елементів контейнера слід дуже обережно (і взагалі, не бажано).

## Винятки

В бібліотеці STL є лише одна функція, яка генерує виняток, – метод `at()`, який реалізує доступ до елемента вектору чи двосторонньої черги за його індексом. Функція генерує стандартний виняток `out_of_range`, якщо аргумент перевищує `size()`. В інших випадках генерується стандартний виняток `bad_alloc` при нестачі пам'яті.

STL надає базову гарантію безпеки винятків: виникнення винятків в STL не приводить до втрати ресурсів та порушення контейнерних інваріантів. Це означає, що якщо програміст акуратно створює свої класи, враховуючи потенційні винятки, то STL гарантує безпеку опрацювання елементів в контейнері.

## Ітератори

Ітератори STL забезпечують доступ до елементів контейнера. Всі контейнери надають методи для присвоєння початкового та кінцевого значення ітераторам. Ітератори контейнерів реалізовані як вкладені класи, тому при діях з контейнерами можна використовувати і відповідні ітератори. Наприклад, при роботі з контейнером-вектором

```
vector<int> L;
```

можна оголосити в програмі і відповідний ітератор:

```
vector<int>::iterator il = L.begin();
```

## Категорії ітераторів

Ітератори STL поділені на категорії. В STL визначено 5 категорій ітераторів – в порядку збільшення «можливостей»:

- вхідний (input) – зчитування елементів контейнера в прямому порядку;
- вихідний (output) – запис елементів контейнера в прямому порядку;
- прямий (forward) – зчитування та запис елементів контейнера в прямому порядку;
- двосторонній (bidirectional) – зчитування та запис елементів контейнера в прямому та зворотному порядках;
- довільного доступу (random access) – довільний доступ для зчитування та запису елементів контейнера.

На практиці для будь-якого контейнера гарантується двосторонній ітератор. Крім того, для всіх контейнерів, крім списку, забезпечується ітератор довільного доступу, який реалізує повну арифметику вбудованих вказівників.

Ітератори можуть бути константними та неконстантними. Константні ітератори не використовуються для зміни елементів контейнера.

Ітератори можуть бути дійсними (коли ітератору відповідає елемент контейнера) і недійсними (при відсутності відповідного елемента контейнера) – мова не йде про тип `float` чи `double`! Ітератор може бути недійсним із-за трьох причин:

- ітератор не був ініціалізований;
- ітератор дорівнює `end()`;
- контейнер, з яким пов'язаний ітератор, змінив розміри чи взагалі був знищений.

Останнє часто стає джерелом помилок в програмах, бо програмісти забувають, що **при видаленні чи вставці елемента ітератори стають недійсними**.

Для всіх категорій ітераторів визначені наступні операції (*i* та *j* – ітератори):

- `i++` – зміщення вперед (повертає стару позицію);
- `++i` – зміщення вперед (повертає нову позицію);
- `i = j` – присвоєння ітераторів;
- `i == j` – порівняння ітераторів на збіг;
- `i != j` – порівняння ітераторів на відмінність;
- `*i` – звертання до елемента;
- `i->member` – звертання до поля/метода елемента; еквівалентно `(*i).member`.

Інші операції, визначені для різних категорій ітераторів, показані в наступній таблиці.

**Операції з ітераторами різних категорій**

Категорія ітератора	Операції	Контейнери
Вхідний	<code>x = *i</code> – зчитування елемента	всі
Вихідний	<code>*i = x</code> – запис елемента	всі
Прямий	<code>x = *i, *i = x</code>	всі
Двосторонній	<code>x = *i, *i = x, --i, i--</code>	всі
Довільного доступу	<code>x = *i, *i = x, --i, i--, i + n, i - n, i += n, i -= n, i &lt; j, i &gt; j, i &lt;= j, i &gt;= j</code>	всі, крім <code>list</code>

## Адаптери ітераторів

Адаптери ітераторів – це: зворотні ітератори, ітератори вставки, потокові ітератори.

Методи `rbegin()` та `rend()`, які реалізовані для всіх контейнерів, повертають початкове та кінцеве значення *зворотного ітератора*. Використання зворотного ітератора нічим не відрізняється від використання прямого ітератора, лише операція `++` визначена так,

що переміщення по контейнеру здійснюється від останнього елемента до першого. Зворотні ітератори дозволяють відсортувати контейнер у зворотному порядку.

Для використання *ітераторів вставки* та *потоків ітераторів* слід підключити файл заголовку

```
#include <iterator>
```

Ітератори вставки і потокові ітератори зазвичай використовуються в поєднанні з алгоритмами модифікації. Алгоритми модифікації працюють в режимі заміщення: результуючий контейнер на момент виклику алгоритму має існувати і у ньому має бути достатньо елементів. Попередні значення елементів замінюються новими.

*Ітератори вставки* дозволяють вставляти в існуючий порожній контейнер нові елементи. STL надає три види ітераторів вставки:

- `back_insert_iterator` – вставка в кінець контейнера;
- `front_insert_iterator` – вставка в початок контейнера;
- `insert_iterator` – вставка перед заданим елементом.

Ітератор вставки в кінець (кінцевий ітератор) працює з будь-яким послідовним контейнером. Ітератор вставки в початок (початковий ітератор) неможна використовувати з вектором, бо вектор не забезпечує вставку елементів в початок – для нього не визначено метод `push_front()`. Третій вид ітераторів підходить для будь-якого контейнера.

Безпосередньо звертатися до ітераторів вставки не дуже зручно, тому в бібліотеці реалізовані функції-обгортки у вигляді шаблонів. Зокрема, кінцевий ітератор для контейнера створюється функцією `back_inserter()`:

```
template<class Container>
back_insert_iterator<Container> back_inserter(Container& x);
```

Аналогічно виглядає і функція `front_inserter()`:

```
template<class Container>
front_insert_iterator<Container> front_inserter(Container& x);
```

Обидві функції отримують в якості аргументу контейнер, в який буде виконуватися вставка, і повертають відповідний ітератор вставки.

Нарешті, функція `inserter()` забезпечує роботу з довільним ітератором вставки. В якості аргументів, крім контейнера, передається ще ітератор, який визначає позицію вставки (елемент, перед яким буде виконана вставка):

```
template<class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i);
```

*Потокові ітератори* – це ітератори у вигляді адаптерів, які сприймають потік даних за джерело чи приймач даних. Потіковий ітератор вводу зчитує елементи із потоку даних, а потіковий ітератор виводу – записує дані у потік.

*Потоковий ітератор виводу* створюється одним із конструкторів:

- `ostream_iterator<T>(stream, delim)` – створює ітератор, пов’язаний з потоком виводу `stream`, в якому значення при виводі відокремлюються літерним рядком `delim` типу `const char *`;
- `ostream_iterator<T>(stream)` – створює ітератор, пов’язаний з потоком виводу `stream`, в якому значення при виводі не відокремлюються.

Виведення в потік виконується операцією присвоєння, а переміщення – операцією `++`, як і переміщення ітераторів вставки. Можна використовувати ітератор виводу безпосередньо, проте зазвичай ітератор виводу використовується одночасно з алгоритмами.

*Потоковий ітератор вводу* дозволяє алгоритмам зчитувати дані не з контейнера, а з потоку введення. Для отримання даних з потоку слід мати два ітератори: *потоковий ітератор вводу* та *ітератор кінця потоку*. Ці ітератори створюються конструкторами:

- `istream_iterator<T>(stream)` – створює ітератор, пов’язаний з потоком вводу `stream`;
- `istream_iterator<T>()` – створює ітератор кінця потоку.

Якщо спроба зчитування – невдала (наприклад, виникає ситуація `end-of-file`), то потоковий ітератор вводу перетворюється в ітератор кінця потоку.

Ітератори вводу можна визначати окремими змінними, а можна використовувати анонімні ітератори безпосередньо як аргументи при виклику алгоритмів.

## Допоміжні функції для ітераторів

Для більшої зручності STL надає функції для роботи з ітераторами:

```
// перемістити ітератор
template<class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);

// обчислити відстань між ітераторами
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Щоб використовувати ці функції, слід підключити файл

```
#include <iterator>
```

Функція `advance()` дозволяє виконати операції `i += n` та `i -= n` з ітератором будь-якої категорії, а не лише з ітератором довільного доступу.

Функція `distance()` обчислює відстань між двома ітераторами, які посилаються на елементи одного контейнера. Відстань – це значення типу `difference_type`, яке є різницею `last - first`, і фактично визначає кількість елементів у цьому діапазоні.

# Алгоритми

Файл заголовку `<algorithm>` визначає набір функцій, спеціально розроблених для використання на діапазонах елементів.

Діапазон – це будь-яка послідовність об'єктів, до якої можна отримати доступ за допомогою ітераторів або вказівників, таких як масив чи екземпляр деяких контейнерів STL.

**Зауваження:** алгоритми працюють за допомогою ітераторів безпосередньо над значеннями, не впливаючи жодним чином на структуру будь-якого можливого контейнера (це ніколи не впливає на розмір або розподіл пам'яті для контейнера).

Всі функції в бібліотеці `<algorithm>` поділяються на наступні категорії:

- Ті, які не модифікують послідовність
- Ті, які модифікують послідовність
- Розбиття
- Сортвання
- Бінарний пошук
- Злиття
- Максимальна купа
- Min / max
- Лексикографічні операції

# Теоретичні відомості

## Нові можливості C++

В цьому розділі розглядаються нові можливості C++, які будуть використовуватися в прикладах з теми «STL – стандартна бібліотека шаблонів».

### Автоматичне виведення типу

`auto` – виводить тип оголошеної змінної з виразу ініціалізації.

Стандарт C++ визначає початкове і змінене значення для цього ключового слова. До Visual Studio 2010 ключове слово `auto` оголошує змінну в автоматичному класі зберігання, тобто змінну з локальним часом існування. Починаючи з Visual Studio 2010 ключове слово `auto` оголошує змінну, тип якої виведений з виразу ініціалізації в його оголошенні.

### Синтаксис

`auto` *декларатор ініціалізатор*;

### Пояснення

Ключове слово `auto` скеровує компілятор використовувати вираз ініціалізації оголошеної змінної або параметра лямбда-виразу, щоб вивести його тип.

Рекомендується використовувати ключове слово `auto` для більшості ситуацій, якщо тільки не потрібно перетворення типу.

Щоб використовувати ключове слово `auto`, слід вказати його замість типу для оголошення змінної, а потім – вказати вираз ініціалізації. Крім того, можна змінити ключове слово `auto` за допомогою модифікаторів і деклараторів, таких як `const`, `volatile`, вказівник (\*), посилання (&) і *rvalue*-посилання (&&). Компілятор обчислює вираз ініціалізації, а потім використовує ці відомості, щоб вивести тип змінної.

Вираз ініціалізації `auto` може приймати кілька форм:

- Синтаксис універсальної ініціалізації, наприклад `auto a {42};`.
- Синтаксис присвоєння, наприклад `auto b = 0;`.
- Синтаксис універсального присвоєння, який об'єднує дві попередні форми, такі як `auto c = {3.14156};`.
- Пряма ініціалізація або синтаксис в стилі конструктора, наприклад `auto d (1.41421f);`.

Якщо `auto` використовується для оголошення параметра в операторі циклу `for` на основі діапазону, використовується інший синтаксис ініціалізації, наприклад:

```
for (auto& i: iterable) do_action(i);
```

Ключове слово `auto` є заповнювачем для типу, але не є типом. Тому ключове слово `auto` не може використовуватися в операціях приведення типу або операторах, таких як `sizeof` і `typeid`.

## Приклади

Приклад 1.

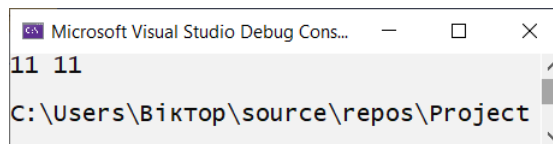
```
#include <iostream>

using namespace std;

int main()
{
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```



В цьому прикладі `myAuto` – це `int`, а НЕ посилання на `int`, тому результатом буде `11 11`, а не `11 12`.

Приклад 2.

```
#include <initializer_list>

int main()
{
    auto A = { 1, 2 }; // std::initializer_list<int>
    auto B = { 3 };    // std::initializer_list<int>
    auto C{ 4 };       // int

    // E1587: cannot deduce 'auto' type
    auto D = { 5, 6.7 };

    // E2663: braced initialization of a variable declared with a placeholder
    //          type but without '=' requires exactly one element inside the braces
    auto E{ 8, 9 };

    return 0;
}
```



### Приклад 3. Наступні команди – еквівалентні:

```
#include <string>
#include <map>
#include <list>

using namespace std;

int main()
{
    // (1)
    int j = 0; // (a) - Variable j is explicitly type int.
    auto k = 0; // (b) - Variable k is implicitly type int because 0 is an integer.

    // (2)
    map<int, list<string>> m;
    map<int, list<string>>::iterator i = m.begin(); // (a)
    auto i = m.begin();                          // (b)

    return 0;
}
```

### Цикл **for** на основі діапазону

```
for (елемент : контейнер)
    команда;
```

Циклічно і послідовно виконує *команду* для кожного *елемента* з *контейнера*.

В якості *команди* може бути блок: { ... }.

Цикл **for** на основі діапазону слід використовувати для створення циклів, які мають виконуватися за допомогою *діапазону елементів контейнера*. Діапазон елементів – це те, що можна перебирати, наприклад, `std::vector` або будь-яка інша послідовність стандартної бібліотеки шаблонів C++, діапазон якої визначається ітераторами `begin()` та `end()`. Ім'я, оголошене перед двокрапкою, є локальним для команди **for** і не може бути повторно оголошене в заголовку циклу **for** або в *команді*. Зазвичай для оголошення *елемента* використовується ключове слово **auto**.

- Такі цикли автоматично розпізнають масиви.
- Такі цикли автоматично розпізнають контейнери з методами `begin()` і `end()`.
- Для всіх інших ітераторів в них використовуються пошук, що залежить від аргументів (`begin()` і `end()`).

Цикл **for** на основі діапазону завершується, коли виконується одна з цих команд: **break**, **return** або перехід **goto** в позначену команду за межами циклу **for**, оснований на діапазоні. Команда **continue** в циклі **for** на основі діапазону завершує тільки поточну ітерацію.

Приклад:

```
// range-based-for.cpp
// compile by using: cl /EHsc /nologo /W4
#include <iostream>
```

```

#include <vector>
using namespace std;

int main()
{
    // Basic 10-element integer array.
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Range-based for loop to iterate through the array.
    for (int y : x) { // Access by value using a copy declared as a specific type.
        // Not preferred.
        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.

    for (auto y : x) { // Copy of 'x', almost always undesirable
        cout << y << " ";
    }
    cout << endl;

    for (auto& y : x) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is needed.
        cout << y << " ";
    }
    cout << endl;

    for (const auto& y : x) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-place.
    for (const auto& j : v) {
        cout << j << " ";
    }
    cout << endl;
    cout << "end of vector test" << endl;
}

```

```

Microsoft Visual Studio Debug Console
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
end of integer array test

0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159 9.14159
end of vector test

C:\Users\Biktop\source\repos\Project5\Debug\Project5.exe (process 5208) exited with
code 0.

```

## Лямбда-вирази

Лямбда-вираз (або просто «лямбда») дозволяє визначити анонімну функцію всередині іншої функції.

Для C++11 і пізніших версій, лямбда-вираз – це зручний спосіб визначити анонімну функцію безпосередньо в місці, де вона викликається.

Ця можливість є дуже важливою перевагою, оскільки дозволяє уникати як захаращення простору імен зайвими об'єктами, так і визначити функцію якомога ближче до місця її першого використання.

Приклад:

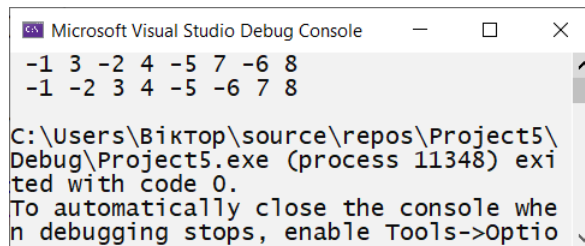
```
#include <algorithm>
#include <cmath>
#include <iostream>

void abs_sort(int* x, unsigned n)
{
    std::sort(x, x + n,
        // Lambda expression begins
        [](int a, int b) {
            return (std::abs(a) < std::abs(b));
        } // end of lambda expression
    );
}

int main()
{
    int a[] = { -1, 3, -2, 4, -5, 7, -6, 8 };
    for (auto e : a)
        std::cout << ' ' << e;
    std::cout << '\n';

    abs_sort(a, 8);
    for (auto e : a)
        std::cout << ' ' << e;
    std::cout << '\n';

    return 0;
}
```



Лямбда-вирази мають наступний синтаксис:

```
[секція_фіксації](список_параметрів) -> тип_результату
{
    тіло_лямбда-виразу;
}
```

Поля *секція\_фіксації* і *параметри* можуть бути порожніми, якщо вони не потрібні програмісту.

Поле *тип\_результату* є не обов'язковим – якщо його немає, то буде використовуватися виведення типу за допомогою ключового слова **auto**.

Лямбда-вирази не мають імен.

Найпростіший лямбда-вираз може мати наступний вигляд:

```
[]() {};
```

// визначили лямбда-вираз без секції фіксації, параметрів,  
// типу результату і команд

## Секція фіксації

Лямбда-вираз може мати свої локальні змінні (в C++14), а також отримувати доступ до змінних з навколишньої області – «захоплювати» їх. Лямбда-вираз починається з секції фіксації, яка вказує, які саме змінні захоплюються і чи захоплення є значенням, чи – посиланням. Доступ до змінних з префіксом з амперсандом (&) здійснюється за посиланням, а до змінних без префікса – за значенням.

Порожня секція фіксації ([]) показує, що тіло лямбда-виразу не здійснює доступ до змінних у зовнішній області видимості.

Можна використовувати режим захоплення за умовчанням, щоб вказати, як захоплювати всі зовнішні змінні, які використовуються в лямбда-виразі:

- [&] – означає, що всі зовнішні змінні, які використовуються в лямбда-виразі, захоплюються за посиланням,
- [=] – означає, вони захоплюються за значенням.

## Список параметрів

Крім можливості фіксації змінних, лямбда-вирази можуть приймати вхідні параметри. Список параметрів є необов'язковим і в більшості випадків нагадує список параметрів для функції.

Приклад:

```
auto y = [](int first, int second)
{
    return first + second;
};
```

## Тип результату

Тип результату лямбда-виразу виводиться автоматично. Не обов'язково використовувати ключове слово **auto**, якщо не вказано тип результату. Тип результату лямбда-виразу нагадує деякі типи результатів в звичайному методі або функції. Тип результату лямбда-виразу має слідувати за списком параметрів і знаком «стрілочка» ->.

Можна не вказувати тип результату лямбда-виразу, якщо тіло лямбда-виразу містить лише одну команду `return` або лямбда-вираз не повертає значення. Якщо тіло лямбда-виразу містить лише одну команду `return`, компілятор виводить тип результату з типу виразу в команді `return`. Якщо лямбда-вираз не повертає значення, то компілятор виводить тип результату як `void`.

Приклад:

```
int main()
{
    auto x1 = [](int i) { return i; }; // OK: return type is int

    auto x2 = [] { return{ 1, 2 }; }; // E2366: a brace-enclosed list does not provide
                                     // a return type for this lambda

    return 0;
}
```

## Тіло лямбда-виразу

Тіло лямбда-виразу (блок команд) може містити все, що може містити тіло звичайного методу або функції. Тіло лямбда-виразу, як і тіло звичайної функції, може здійснювати доступ до наступних типів змінних:

- фіксовані змінні із зовнішнього області видимості (див. Секція фіксації);
- параметри;
- локально оголошені змінні;
- поля класу, коли лямбда-вираз оголошений всередині класу, і захоплено вказівник `this`;
- будь-яка змінна, яка має статичну тривалість зберігання (наприклад, глобальна змінна).

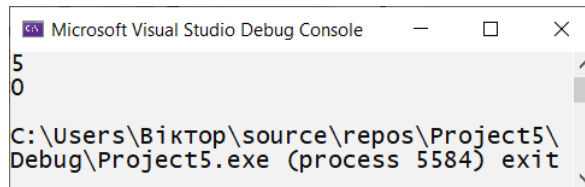
У наступному прикладі міститься лямбда-вираз, яке явно фіксує змінну `n` за значенням і неявно фіксує змінну `m` за посиланням.

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;

    [&, n](int a) mutable { m = ++n + a; }(4);

    cout << m << endl << n << endl;
}
```



```
Microsoft Visual Studio Debug Console
5
0
C:\Users\Віктор\source\repos\Project5\
Debug\Project5.exe (process 5584) exit
```

Оскільки змінна `n` фіксується за значенням, її значення після виклику лямбда-виразу залишається рівним 0. Специфікація `mutable` означає, що змінну `n` можна змінити в лямбда-виразі.

## Приклади

Без лямбда-виразу:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string>

// static в цьому контексті означає внутрішнє зв'язування
static bool containsNut(std::string str)
{
    // Функція std::string::find() повертає std::string::npos,
    // якщо вона не знайшла підрядка.
    // В іншому випадку, вона повертає індекс,
    // для якого відбувається входження підрядка в рядок str

    return (str.find("nut") != std::string::npos);
}

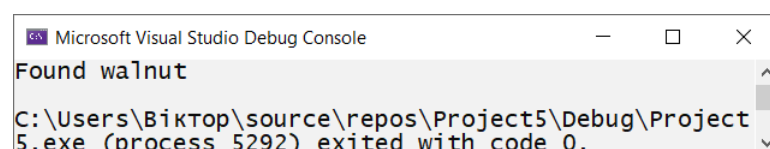
int main()
{
    std::array<std::string, 4> arr{ "apple", "banana", "walnut", "lemon" };

    // std::find_if() приймає вказівник на функцію
    auto found{ std::find_if(arr.begin(), arr.end(), containsNut) };

    if (found == arr.end())
    {
        std::cout << "No nuts\n";
    }
    else
    {
        std::cout << "Found " << *found << '\n';
    }

    return 0;
}
```

Результат виконання:



```
Microsoft Visual Studio Debug Console
Found walnut
C:\Users\Віктор\source\repos\Project5\Debug\Project
5.exe (process 5292) exited with code 0.
```

З лямбда-виразом:

```
#include <algorithm>
#include <array>
#include <iostream>
#include <string>


int main()
{
    std::array<std::string, 4> arr{ "apple", "banana", "walnut", "lemon" };

    // Визначаємо функцію безпосередньо в тому місці, де збираємося її використовувати
    auto found{ std::find_if(arr.begin(), arr.end(),
        [](std::string str) // ось лямбда-вираз,
        {                  // без секції фіксації
            return (str.find("nut") != std::string::npos);
        }) };

    if (found == arr.end())
    {
        std::cout << "No nuts\n";
    }
    else
    {
        std::cout << "Found " << *found << '\n';
    }

    return 0;
}
```

Результат виконання – той самий:



```
Microsoft Visual Studio Debug Console
Found walnut
C:\Users\Віктор\source\repos\Project5\Debug\Project
5.exe (process 5292) exited with code 0.
```

## STL – стандартна бібліотека шаблонів

В цій темі розглядаються основи стандартної бібліотеки шаблонів (STL – standard template library). Розглядаються послідовні та асоціативні контейнери, ітератори, стандартні алгоритми та функтори.

Основні складові стандартної бібліотеки шаблонів – це *контейнери*, *ітератори* та *алгоритми*. Основна концепція STL – відокремлення даних від операцій. Дані зберігаються в контейнерах, операції визначаються адаптованими алгоритмами. Ітератори поєднують ці два компоненти, – завдяки ним будь-який алгоритм може працювати майже з будь-яким контейнером. Універсальність алгоритмів стає ще більшою при використанні *функторів*, які реалізовані в STL.

Важливою особливістю STL є те, що всі компоненти працюють з довільними типами, – бо вони оформлені у вигляді шаблонів.

### Контейнери

Бібліотека контейнерів – це загальна колекція шаблонів класів та алгоритмів, які дозволяють програмістам легко реалізовувати загальні структури даних, такі як черги, списки та стеки. Існує три види контейнерів – *послідовні контейнери*, *асоціативні контейнери* та *невпорядковані асоціативні контейнери* – кожен з яких призначений для підтримки різного набору операцій.

Контейнер керує пам'яттю, що виділяється для його елементів, і надає методи для доступу до них – або безпосередньо, або через ітератори (об'єкти із властивостями, подібними до вказівників).

Більшість контейнерів мають принаймні кілька спільних методів і мають спільну функціональність. Який контейнер є найкращим для конкретного додатку, залежить не тільки від його функціональності, але й від його ефективності при різних робочих навантаженнях.

*Контейнер STL* – це набір однотипних елементів. Всі контейнери поділяються на наступні види – послідовні, асоціативні та неспорядковані асоціативні контейнери:

- *Послідовні контейнери* – це неспорядковані колекції однотипних елементів. Кожний елемент розташований в контейнері у певній позиції, яка залежить лише від часу та місця вставки, і не залежить від значення елемента. До послідовних контейнерів відносяться вектор (*vector*), список (*list*) і двостороння черга (*deque*).



- *Асоціативні контейнери* – це впорядковані за умовчанням колекції однотипних елементів, в яких позиція елемента залежить від його значення. До асоціативних контейнерів належать множина (`set`), мультимножина (`multiset` – множина з елементами, які повторюються), відображення (`map`) та мультिवідображення (`multimap` – відображення з елементами, які повторюються).
- *Невпорядковані асоціативні контейнери* – добавлені в нових версіях C++. Це `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`.

Крім вище зазначених основних контейнерів, STL ще містить набір спеціальних контейнерів-адаптерів: стек (`stack`), черга (`queue`) та черга з пріоритетами (`priority_queue`).

В бібліотеці STL реалізовані інші структури, які формально не є стандартними контейнерами елементів: булевий вектор (`vector<bool>`) та бітові поля (`bitset`).

**Зауваження:** STL в різних системах програмування може містити інші, нестандартні, контейнери.

Всі контейнери – динамічні: один із параметрів шаблону визначає розподіл пам'яті.

Для використання контейнерів та адаптерів слід підключити файли заголовків:

```
#include <vector>    // контейнер-вектор та vector<bool>
#include <deque>     // контейнер-двостороння черга
#include <list>      // контейнер-список
#include <stack>     // стек
#include <queue>     // черга та черга з пріоритетами
#include <set>       // множина та мультимножина
#include <map>       // відображення та мультिवідображення
#include <bitset>    // послідовність бітів
```

## Вимоги до елементів контейнера

Елементи STL-контейнера можуть бути довільного типу `T`, але цей тип має задовольняти вимогам:

1. Всі контейнери створюють внутрішні копії своїх елементів та повертають ці копії. Тому `T` має мати конструктор копіювання. Копія має дорівнювати оригіналу, і поведінка копії не має відрізнятися від поведінки оригіналу.
2. Контейнери використовують операцію присвоєння для заміни старих елементів новими. Це означає, що тип (клас) `T` має підтримувати операцію присвоєння.
3. Контейнери знищують свої внутрішні копії при видаленні елементів із контейнера. Тому деструктор `T` не може бути закритим (`private`).

Ці три операції (копіювання, присвоєння, видалення) автоматично генеруються для будь-якого класу, якщо відсутні їх явні визначення. Тому будь-який клас за умовчанням

задовольняє цим вимогам, якщо для нього не перевизначати цих операцій. Цим вимогам також задовольняють всі вбудовані типи.

Проте цим вимогам не задовольняють елементи бітових полів та булевих векторів – тому вони не вважаються стандартними контейнерами.

Стандартні вбудовані вказівники теж не повністю задовольняють зазначеним вимогам, тому використовувати вказівники в якості елементів контейнера слід дуже обережно (і взагалі, не бажано).

## Винятки

В бібліотеці STL є лише одна функція, яка генерує виняток, – метод `at()`, який реалізує доступ до елемента вектору чи двосторонньої черги за його індексом. Функція генерує стандартний виняток `out_of_range`, якщо аргумент перевищує `size()`. В інших випадках генерується стандартний виняток `bad_alloc` при нестачі пам'яті.

STL надає базову гарантію безпеки винятків: виникнення винятків в STL не приводить до втрати ресурсів та порушення контейнерних інваріантів. Це означає, що якщо програміст акуратно створює свої класи, враховуючи потенційні винятки, то STL гарантує безпеку опрацювання елементів в контейнері.

## Послідовні контейнери

*Послідовні контейнери* реалізують структури даних, до яких можна отримати послідовний доступ.

<code>array</code> (C++11)	статичний суміжний (неперервний) масив (шаблон класу)
<code>vector</code>	динамічний суміжний (неперервний) масив (шаблон класу)
<code>deque</code>	двостороння черга (шаблон класу)
<code>forward_list</code> (C++11)	однозв'язний список (шаблон класу)
<code>list</code>	двозв'язний список (шаблон класу)

Всі *послідовні контейнери* мають подібний інтерфейс. Класи-шаблони мають однакові параметри:

```
template <class T, class Allocator = allocator<T>>
class vector
{ /* ... */ };

template <class T, class Allocator = allocator<T>>
class deque
{ /* ... */ };
```

```
template <class T, class Allocator = allocator<T>>
class list
{ /* ... */ };
```

Параметр `T` визначає тип елементів контейнера.

Параметр `Allocator` задає розподільувач пам'яті. Для всіх контейнерів реалізовані стандартні розподільувачі за умовчанням. Цей параметр використовується у двох випадках: коли створюється свій розподільувач пам'яті і коли послідовний контейнер використовується в якості аргументу іншого класу-шаблону.

На початку кожного послідовного контейнера визначено типи:

```
typedef T value_type;
typedef Allocator allocator_type;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
class iterator;
class const_iterator;
typedef typename std::reverse_iterator<iterator> reverse_iterator;
typedef typename std::reverse_iterator<const iterator> const_reverse_iterator;
```

Конструктори дозволяють створювати послідовні контейнери наступними способами (замість `container` слід підставити `vector`, `list` або `deque`, а замість `type` – довільний тип):

```
container<type> v;           // порожній контейнер, який не містить елементів
container<type> v(n);        // контейнер із n елементів
container<type> v(n, val);    // контейнер із n елементів, які дорівнюють val

container<type>::iterator beg = v.begin();
container<type>::iterator end = v.end();

container<type> v1(beg, end); // контейнер із n елементів, ініціалізованих
                             // елементами з інтервалу ітераторів [beg, end)

container<type> v2(v1);       // контейнер v2 - копія контейнера v1
```

В першому випадку виділяється пам'ять, але елементів в контейнері немає.

У другому та третьому випадку задано початкову кількість елементів. Кількість елементів `n` можна задавати не лише константою, а довільним виразом, який обчислюється при виконанні програми. Якщо значення ініціалізації `val` не вказано, то виконується ініціалізація нулями.

Конструктор з аргументами-ітераторами дозволяє створювати контейнер із будь-якого іншого контейнера з елементами того ж самого типу. В якості джерела даних можна використовувати стандартний масив – тоді аргументами будуть вказівники на його елементи:

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
std::vector<int> b(a, a + 5);           // скопіювали 5 елементів із a до b
std::for_each(b.cbegin(), b.cend(), print); // 1 2 3 4 5
```

Будь-який стандартний контейнер має конструктор копіювання.

Всі послідовні контейнери мають методи для визначення та зміни розміру контейнера.

## array

Визначається у файлі заголовку <array>.

```
template <class T, size_t N>
class array;
```

std::array є контейнером, який інкапсулює масиви фіксованого розміру.

Цей контейнер є агрегованим типом з тією ж семантикою, що і структура, що містить масив у стилі C.

T[N] – єдине нестатичне поле.

На відміну від масиву в стилі C, він автоматично не приводиться до типу T\*.

Як агрегатний тип, він може бути ініціалізований за допомогою агрегатної ініціалізації: std::array<int, 3> a = { 1, 2, 3 };.

Структура поєднує продуктивність і доступність масиву в стилі C з перевагами стандартного контейнера, такими як знання власного розміру, підтримка присвоєння, ітератори довільного доступу тощо.

Для випадку нульової довжини (N == 0): array.begin() == array.end().

array може використовуватися як набір N елементів одного типу.

### Приклад 1.

```
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    // construction uses aggregate initialization
    std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 prior to
                                     // the CWG 1270 revision (not needed in C++11
                                     // after the revision and in C++14 and beyond)

    std::array<int, 3> a2 = { 1, 2, 3 }; // double braces never required after =

    std::array<std::string, 2> a3 = { std::string("a"), "b" };

    // container operations are supported
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(),
        std::ostream_iterator<int>(std::cout, " "));

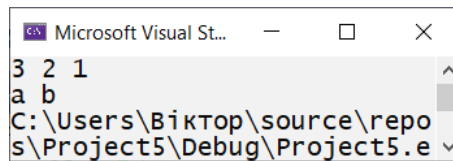
    std::cout << '\n';

    // ranged for loop is supported
    for (const auto& s : a3)
        std::cout << s << ' ';
```

```

// deduction guide for array creation (since C++17)
// std::array a4{ 3.0, 1.0, 4.0 }; // -> std::array<double, 3>
std::array<double, 3>{ 3.0, 1.0, 4.0 };
}

```



## vector

Визначається у файлі заголовку <vector>.

```

template <class T, class Alloc = std::allocator<T>>
class vector; // generic template

```

Елементи зберігаються суміжно (неперервно), а це означає, що вони доступні не лише за допомогою ітераторів, але й за допомогою зміщень для звичайних вказівників на елементи. Це означає, що вказівник на елемент вектора може бути переданий будь-якій функції, яка очікує вказівника на елемент масиву.

Пам'ять для вектора опрацьовується автоматично, при необхідності розширюється та скорочується. Вектори зазвичай займають більше місця, ніж статичні масиви, оскільки більше пам'яті виділяється для опрацювання майбутнього зростання. Таким чином, пам'ять для вектора не потрібно перерозподіляти кожен раз, коли вставляється елемент, а лише тоді, коли додаткова пам'ять вичерпується. Загальний обсяг виділеної пам'яті можна отримати за допомогою функції `capacity()`. Додаткову пам'ять можна повернути системі за допомогою виклику `shrink_to_fit()`. (з C++ 11)

Перерозподіл – це, як правило, дорогі операції з точки зору продуктивності. Функцію `reserve()` можна використовувати для усунення перерозподілу, якщо кількість елементів відома заздалегідь.

### Приклад 1.

```

#include <iostream>
#include <vector>

int main()
{
    // Create a vector containing integers
    std::vector<int> v = { 7, 5, 16, 8 };

    // Add two more integers to vector
    v.push_back(25);
    v.push_back(13);

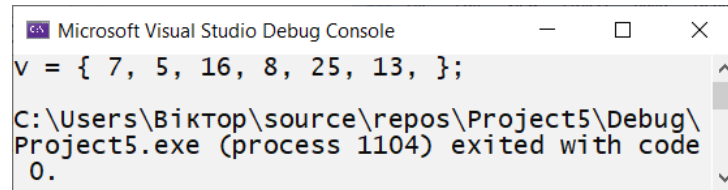
    // Print out the vector
    std::cout << "v = { ";
    for (int n : v)
    {

```

```

        std::cout << n << ", ";
    }
    std::cout << "}; \n";
}

```



## Приклад 2.

```

// std::begin / std::end example
#include <iostream>      // std::cout
#include <vector>        // std::vector, std::begin, std::end

int main()
{
    int foo[] = { 10,20,30,40,50 };
    std::vector<int> bar;

    // iterate foo: inserting into bar
    for (auto it = std::begin(foo); it != std::end(foo); ++it)
        bar.push_back(*it);

    // iterate bar: print contents:
    std::cout << "bar contains:";
    for (auto it = std::begin(bar); it != std::end(bar); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // Output:
    // bar contains: 10 20 30 40 50

    return 0;
}

```

## deque

Визначається у файлі заголовку <deque>.

```

template <class T, class Alloc = std::allocator<T>>
class deque;

```

`std::deque` (двостороння черга) — це індексований послідовний контейнер, який дозволяє швидко вставляти та видаляти елементи як на початку, так і в кінці контейнера. Крім того, вставка та видалення на кожному кінці `deque` ніколи не робить недійсними вказівники або посилання на решту елементів.

На відміну від `std::vector`, елементи `deque` не зберігаються суміжно.

Пам'ять для `deque` автоматично розширюється і за необхідності скорочується. Розширення `deque` дешевше, ніж розширення `std::vector`, оскільки воно не передбачає копіювання існуючих елементів у нове місце пам'яті. З іншого боку, `deque` зазвичай мають великі мінімальні витрати на пам'ять.

Приклад:

```

#include <iostream>
#include <deque>

int main()
{
    // Create a deque containing integers
    std::deque<int> d = { 7, 5, 16, 8 };

    // Add an integer to the beginning and end of the deque
    d.push_front(13);
    d.push_back(25);

    // Iterate and print values of deque
    for (int n : d)
    {
        std::cout << n << '\n';
    }
}

```

```

Micros...
13
7
5
16
8
25
C:\Users\Віктор\source\repos\Project5\Debug\Project5.exe (pr

```

## forward\_list

Визначається у файлі заголовку <forward\_list>.

```

template <class T, class Alloc = std::allocator<T>>
class forward_list;

```

`std::forward_list` є контейнером, який підтримує швидке вставлення та видалення елементів з будь-якої точки контейнера. Швидкий довільний доступ не підтримується. Він реалізований як однозв'язний список. У порівнянні зі `std::list` цей контейнер забезпечує більш ефективно зберігання місця, коли двонаправлена ітерація не потрібна.

Додавання, видалення та переміщення елементів у списку або в декількох списках не робить ітератори, які в цей час посилаються на інші елементи у списку, недійсними. Однак ітератор або посилання, що посилаються на елемент, втрачають силу, коли відповідний елемент вибуває (через `erase_after`) зі списку.

## list

Визначається у файлі заголовку <list>.

```

template <class T, class Alloc = std::allocator<T>>
class list;

```

`std::list` є контейнером, який підтримує вставку та видалення елементів з будь-якої точки контейнера. Швидкий довільний доступ не підтримується. Зазвичай він реалізується як подвійно пов'язаний список. Порівняно зі `std::forward_list`, цей контейнер надає можливість двонаправленої ітерації, але є менш ефективним щодо розподілу пам'яті.

Додавання, видалення та переміщення елементів у списку або в декількох списках не робить ітератори або посилання недійсними. Ітератор стає недійсним лише тоді, коли відповідний елемент буде видалено.

Приклад:

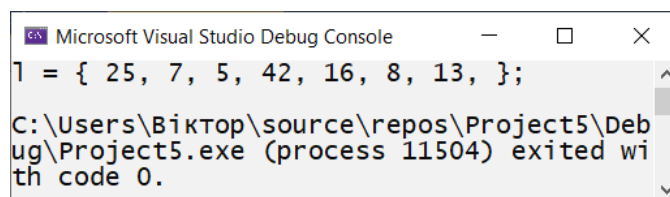
```
#include <algorithm>
#include <iostream>
#include <list>

int main()
{
    // Create a list containing integers
    std::list<int> l = { 7, 5, 16, 8 };

    // Add an integer to the front of the list
    l.push_front(25);
    // Add an integer to the back of the list
    l.push_back(13);

    // Insert an integer before 16 by searching
    auto it = std::find(l.begin(), l.end(), 16);
    if (it != l.end()) {
        l.insert(it, 42);
    }

    // Print out the list
    std::cout << "l = { ";
    for (int n : l) {
        std::cout << n << ", ";
    }
    std::cout << "};\n";
}
```



```
Microsoft Visual Studio Debug Console
l = { 25, 7, 5, 42, 16, 8, 13, };
C:\Users\Biktop\source\repos\Project5\Debug\Project5.exe (process 11504) exited with code 0.
```



## Асоціативні контейнери

Асоціативні контейнери реалізують відсортовані структури даних, які забезпечують швидкий пошук (складність  $O(\log n)$ ).

set	множина – колекція унікальних ключів, відсортованих за ключами (шаблон класу)
map	відображення – колекція пар ключ-значення, відсортованих за ключами, ключі унікальні (шаблон класу)
multiset	мультимножина – колекція ключів, відсортованих за ключами (шаблон класу)
multimap	мультивідображення – колекція пар ключ-значення, відсортованих за ключами (шаблон класу)

### set

Визначається у файлі заголовку <set>.

```
template < class T,                // set::key_type/value_type
          class Compare = std::less<T>, // set::key_compare/value_compare
          class Alloc = std::allocator<T> // set::allocator_type
> class set;
```

`std::set` є асоціативним контейнером, що містить відсортований набір унікальних об'єктів типу `key`. Сортування здійснюється за допомогою функції порівняння ключів `Compare`. Операції пошуку, видалення та вставки мають логарифмічну складність. Набори зазвичай реалізуються як червоно-чорні дерева.

Скрізь, де стандартна бібліотека використовує вимоги `Compare`, унікальність визначається за допомогою відношення еквівалентності. Зокрема, два об'єкти `a` і `b` вважаються еквівалентними, якщо один не відрізняється від іншого: `!comp(a, b) && !comp(b, a)`.

`std::set` має два корисні методи: `std::set::erase` та `std::set::insert`.

### erase

```
// (1)
void erase(iterator position);

// (2)
size_type erase(const value_type& val);

// (3)
void erase(iterator first, iterator last);
```

Стерти елементи.

Видаляє із вказаного контейнера-множини або окремих елементів, або діапазон елементів `[first, last)`. Ефективно зменшує розмір контейнера-множини на кількість вилучених елементів, які знищуються.

Приклад:

```
// erasing from set
#include <iostream>
#include <set>

int main()
{
    std::set<int> myset;
    std::set<int>::iterator it;

    // insert some values:
    for (int i = 1; i < 10; i++) myset.insert(i * 10); // 10 20 30 40 50 60 70 80 90

    it = myset.begin();
    ++it; // "it" points now to 20

    myset.erase(it);

    myset.erase(40);

    it = myset.find(60);
    myset.erase(it, myset.end());

    std::cout << "myset contains:";
    for (it = myset.begin(); it != myset.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    // myset contains: 10 30 50

    return 0;
}
```

## insert

```
// single element (1)
std::pair<iterator, bool> insert(const value_type& val);

// with hint (2)
iterator insert(iterator position, const value_type& val);

// range (3)
template <class InputIterator>
void insert(InputIterator first, InputIterator last);
```

Вставити елемент.

Розширює контейнер-множину, вставляючи нові елементи, ефективно збільшуючи розмір контейнера на кількість вставлених елементів.

Оскільки елементи в множині унікальні, операція вставки перевіряє, чи еквівалентний кожен вставлений елемент елементу, що вже знаходиться в контейнері, і якщо так, елемент не вставляється, повертаючи ітератор до цього існуючого елемента (якщо функція повертає значення).

Подібний контейнер, що дозволяє дублювати елементи, — `multiset`.

На внутрішньому рівні, контейнери-множини сортують усі їх елементи за критерієм, визначеним об'єктом предикату порівняння. Елементи завжди вставляються у відповідне положення відповідно до цього впорядкування.

Приклад:

```
// set::insert (C++98)
#include <iostream>
#include <set>

int main()
{
    std::set<int> myset;
    std::set<int>::iterator it;
    std::pair<std::set<int>::iterator, bool> ret;

    // set some initial values:
    for (int i = 1; i <= 5; ++i)
        myset.insert(i * 10);                // set: 10 20 30 40 50

    ret = myset.insert(20);                  // no new element inserted

    if (ret.second == false) it = ret.first; // "it" now points to element 20

    myset.insert(it, 25);                    // max efficiency inserting
    myset.insert(it, 24);                    // max efficiency inserting
    myset.insert(it, 26);                    // no max efficiency inserting

    int myints[] = { 5,10,15 };              // 10 already in set, not inserted
    myset.insert(myints, myints + 3);

    std::cout << "myset contains:";
    for (it = myset.begin(); it != myset.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    // myset contains: 5 10 15 20 24 25 26 30 40 50

    return 0;
}
```

## map

Визначається у файлі заголовку <map>.

```
template < class Key,                                // map::key_type
          class T,                                    // map::mapped_type
          class Compare = std::less<Key>,             // map::key_compare
          class Alloc = std::allocator<pair<const Key, T> > // map::allocator_type
> class map;
```

std::map — це відсортований асоціативний контейнер, який містить пари ключ-значення з унікальними ключами. Ключі сортуються за допомогою функції порівняння Compare. Операції пошуку, видалення та вставки мають логарифмічну складність. Відображення зазвичай реалізуються як червоно-чорні дерева.

Скрізь, де стандартна бібліотека використовує вимоги Compare, унікальність визначається за допомогою відношення еквівалентності. Зазвичай, два об'єкти a і b

вважаються еквівалентними, якщо один не відрізняється від іншого:

```
!comp(a, b) && !comp(b, a).
```

Приклад:

```
#include <iostream>
#include <map>
#include <string>
#include <string_view>

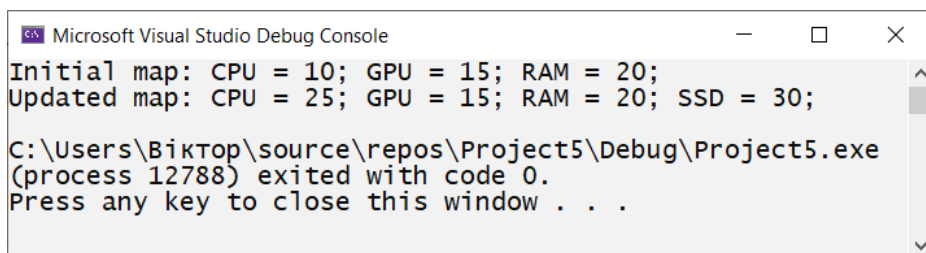
void print_map(std::string comment, const std::map<std::string, int>& m)
{
    std::cout << comment;
    for (const auto& v : m)
    {
        std::cout << v.first << " = " << v.second << "; ";
    }
    std::cout << "\n";
}

int main()
{
    // Create a map of three strings (that map to integers)
    std::map<std::string, int> m{ {"CPU", 10}, {"GPU", 15}, {"RAM", 20}, };

    print_map("Initial map: ", m);

    m["CPU"] = 25; // update an existing value
    m["SSD"] = 30; // insert a new value

    print_map("Updated map: ", m);
}
```



```
Microsoft Visual Studio Debug Console
Initial map: CPU = 10; GPU = 15; RAM = 20;
Updated map: CPU = 25; GPU = 15; RAM = 20; SSD = 30;

C:\Users\Віктор\source\repos\Project5\Debug\Project5.exe
(process 12788) exited with code 0.
Press any key to close this window . . .
```

## Невпорядковані асоціативні контейнери

Невпорядковані асоціативні контейнери реалізують несортовані (хешовані) структури даних, які забезпечують швидкий пошук ( $O(1)$  – в середньому,  $O(n)$  – найгірша складність).

<code>unordered_set</code> (C++11)	колекція унікальних ключів, хешованих за ключами (шаблон класу)
<code>unordered_map</code> (C++11)	колекція пар ключ-значення, хешована ключами, ключі унікальні (шаблон класу)
<code>unordered_multiset</code> (C++11)	набір ключів, хешований за ключами (шаблон класу)
<code>unordered_multimap</code> (C++11)	колекція пар ключ-значення, хешована ключами (шаблон класу)

## Адаптери контейнерів

*Адаптери контейнерів забезпечують різні інтерфейси для послідовних контейнерів.*

<code>stack</code>	адаптує контейнер для забезпечення стека (структура даних LIFO) (шаблон класу)
<code>queue</code>	адаптує контейнер для забезпечення черги (структура даних FIFO) (шаблон класу)
<code>priority_queue</code>	адаптує контейнер для забезпечення черги з пріоритетами (шаблон класу)

### stack

Визначається у файлі заголовку `<stack>`.

```
template <class T, class Container = std::deque<T>>
class stack;
```

Клас `std::stack` – адаптер контейнера, який надає програмісту функціональність стека – структури даних LIFO (останній увійшов, першим вийшов).

Шаблон класу діє як обгортка для базового контейнера – надається лише певний набір функцій. Стек вставляє та видаляє елемент із кінця базового контейнера – `deque`, відомого як вершина стека.

### queue

Визначається у файлі заголовку `<queue>`.

```
template <class T, class Container = std::deque<T>>
class queue;
```

Клас `std::queue` – адаптер контейнера, який надає програмісту функціональність черги – структури даних FIFO (першим увійшов, першим вийшов).

Шаблон класу діє як обгортка для базового контейнера – надається лише певний набір функцій. Черга вставляє елементи в кінець базового контейнера і вилучає їх спочатку.

### priority\_queue

Визначається у файлі заголовку `<queue>`.

```
template <class T, class Container = std::vector<T>,
          class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

Черга з пріоритетами – це адаптер контейнера, який забезпечує постійний час пошуку за рахунок логарифмічної вставки та вилучення.

Заданий користувачем метод `Compare` можна використовувати для зміни порядку, наприклад, використання `std::greater<T>` приведе до того, що найменший елемент відображатиметься як `top()`.

### Приклад:

```
#include <functional>
#include <queue>
#include <vector>
#include <iostream>

template<typename T>
void print_queue(T q) // NB: pass by value so the print uses a copy
{
    while (!q.empty())
    {
        std::cout << q.top() << ' ';
        q.pop();
    }
    std::cout << '\n';
}

int main()
{
    std::priority_queue<int> q;

    const auto data = { 1,8,5,6,3,4,0,9,7,2 };

    for (int n : data)
        q.push(n);

    print_queue(q);

    std::priority_queue<int, std::vector<int>, std::greater<int>>
        q2(data.begin(), data.end());

    print_queue(q2);

    // Using lambda to compare elements.
    auto cmp = [](int left, int right) { return (left ^ 1) < (right ^ 1); };
    std::priority_queue<int, std::vector<int>, decltype(cmp)> q3(cmp);
    // decltype - повертає тип вказаного виразу

    for (int n : data)
        q3.push(n);

    print_queue(q3);
}
```

`auto` і `decltype` зазвичай використовуються для оголошення шаблону функції, тип результату якої залежить від типів аргументів шаблону.

## Втрата ітератора

Методи зчитування ніколи не приводять до втрати ітератора чи посилання. Методи, що модифікують вміст контейнера, можуть привести до втрати ітераторів та / або посилань, як зазначено в цій таблиці.

Категорія	Контейнер	Після <b>вставки</b>		Після <b>видалення</b>		Пояснення
		ітератори дійсні?	посилання дійсні?	ітератори дійсні?	посилання дійсні?	
Послідовні контейнери	<code>array</code>	Не застосовується		Не застосовується		
	<code>vector</code>	Ні		Не застосовується		Вставка змінила ємність
		Так		Так		Перед модифікацією елементів
		Ні		Ні		При або після модифікації елементів
	<code>deque</code>	Ні	Так	Так, крім видалених елементів		Модифікація першого або останнього елемента
			Ні	Ні		Модифікація лише середнього елемента
	<code>list</code>	Так		Так, крім видалених елементів		
Асоціативні контейнери	<code>forward_list</code>	Так		Так, крім видалених елементів		
	<code>set</code> <code>multiset</code> <code>map</code> <code>multimap</code>	Так		Так, крім видалених елементів		

Невпорядковані асоціативні контейнери	<code>unordered_set</code> <code>unordered_multiset</code> <code>unordered_map</code> <code>unordered_multimap</code>	Ні	Так	Не застосовується	Вставка привела до перевпорядкування
		Так		Так, крім видалених елементів	Ніякого перевпорядкування

Тут вставка відноситься до будь-якого методу, який додає один або кілька елементів до контейнера, а видалення – до будь-якого методу, який видаляє один або кілька елементів з контейнера.

- Прикладами методів вставки є `std::set::insert`, `std::map::emplace`, `std::vector::push_back` і `std::deque::push_front`.
  - Зауваження: слід врахувати операцію `std::unordered_map::operator[]`, оскільки вона може вставити елемент у відображення.
- Прикладами методів видалення є `std::set::erase`, `std::vector::pop_back`, `std::deque::pop_front` та `std::map::clear`.
  - `clear` приводить до втрати всіх ітераторів та посилань, оскільки цей метод видаляє всі елементи.

Кінцевий ітератор (`end`) заслуговує на особливу увагу. Взагалі, цей ітератор стає недійсним, як і звичайний ітератор, налаштований на невидалений елемент. Отже, `std::set::end` ніколи не стає недійсним, `std::unordered_set::end` стає недійсним лише при повторному хешуванні, `std::vector::end` завжди стає недійсним (оскільки це завжди відбувається після модифікації елементів) тощо.

Є один виняток: видалення останнього елемента `std::deque` робить кінцевий ітератор недійсним, навіть якщо він налаштований на невидалений елемент контейнера (або будь-який елемент взагалі). Для ітераторів `std::deque` існує лише одна операція модифікації, яка не приводить до втрати ітератора `std::deque::end` – це видалення першого елемента, але не останнього.



## Таблиці версій

	- функції, присутні в C ++ 03
	- функції, присутні з C ++ 11
	- функції, присутні з C ++ 17
	- функції, присутні з C ++ 20

		Послідовні контейнери				
Заголовок		<array>	<vector>	<deque>	<forward_list>	<list>
Контейнер		array	vector	deque	forward_list	list
	(constructor)	(implicit)	vector	deque	forward_list	list
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list
	operator=	(implicit)	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign
Ітератори	begin	begin	begin	begin	begin	begin
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin
	end	end	end	end	end	end
	cend	cend	cend	cend	cend	cend
	rbegin	rbegin	rbegin	rbegin		rbegin
	crbegin	crbegin	crbegin	crbegin		crbegin
	rend	rend	rend	rend		rend
	crend	crend	crend	crend		crend
Доступ до елемента	at	at	at	at		
	operator[]	operator[]	operator[]	operator[]		
	data	data	data			
	front	front	front	front	front	front
	back	back	back	back		back
Ємність	empty	empty	empty	empty	empty	empty
	size	size	size	size		size
	max_size	max_size	max_size	max_size	max_size	max_size
	resize		resize	resize	resize	resize
	capacity		capacity			
	reserve		reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit		
Модифікатори	clear		clear	clear	clear	clear
	insert		insert	insert	insert_after	insert
	insert_or_assign					
	emplace		emplace	emplace	emplace_after	emplace
	emplace_hint					
	try_emplace					
	erase		erase	erase	erase_after	erase
	push_front			push_front	push_front	push_front
	emplace_front			emplace_front	emplace_front	emplace_front
	pop_front			pop_front	pop_front	pop_front
	push_back		push_back	push_back		push_back
	emplace_back		emplace_back	emplace_back		emplace_back
	pop_back		pop_back	pop_back		pop_back
	swap	swap	swap	swap	swap	swap

	merge				merge	merge
	extract					
Операції з контейнером	splice				splice_after	splice
	remove				remove	remove
	remove_if				remove_if	remove_if
	reverse				reverse	reverse
	unique				unique	unique
	sort				sort	sort
Пошук	count					
	find					
	contains					
	lower_bound					
	upper_bound					
	equal_range					
Спостерегачі	key_comp					
	value_comp					
	hash_function					
	key_eq					
Розподільцячі	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator
Контейнер		array	vector	deque	forward_list	list
		Послідовні контейнери				

		Асоціативні контейнери			
Заголовок		<set>		<map>	
Контейнер		set	multiset	map	multimap
	(constructor)	set	multiset	map	multimap
	(destructor)	~set	~multiset	~map	~multimap
	operator=	operator=	operator=	operator=	operator=
	assign				
Ітератори	begin	begin	begin	begin	begin
	cbegin	cbegin	cbegin	cbegin	cbegin
	end	end	end	end	end
	cend	cend	cend	cend	cend
	rbegin	rbegin	rbegin	rbegin	rbegin
	crbegin	crbegin	crbegin	crbegin	crbegin
	rend	rend	rend	rend	rend
	crend	crend	crend	crend	crend
Доступ до елементів	at			at	
	operator[]			operator[]	
	data				
	front				
	back				
Ємність	empty	empty	empty	empty	empty
	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size
	resize				
	capacity				
	reserve				

	shrink_to_fit				
Модифікатори	clear	clear	clear	clear	clear
	insert	insert	insert	insert	insert
	insert_or_assign			insert_or_assign	
	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	try_emplace			try_emplace	
	erase	erase	erase	erase	erase
	push_front				
	emplace_front				
	pop_front				
	push_back				
	emplace_back				
	pop_back				
	swap	swap	swap	swap	swap
	merge	merge	merge	merge	merge
	extract	extract	extract	extract	extract
Операції з контейнером	splice				
	remove				
	remove_if				
	reverse				
	unique				
	sort				
Пошук	count	count	count	count	count
	find	find	find	find	find
	contains	contains	contains	contains	contains
	lower_bound	lower_bound	lower_bound	lower_bound	lower_bound
	upper_bound	upper_bound	upper_bound	upper_bound	upper_bound
	equal_range	equal_range	equal_range	equal_range	equal_range
Спостерегачі	key_comp	key_comp	key_comp	key_comp	key_comp
	value_comp	value_comp	value_comp	value_comp	value_comp
	hash_function				
	key_eq				
Розподільвач	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator
Контейнер		set	multiset	map	multimap
Асоціативні контейнери					

Невпорядковані асоціативні контейнери					
Заголовок		<unordered_set>		<unordered_map>	
Контейнер		unordered_set	unordered_multiset	unordered_map	unordered_multimap
	(constructor)	unordered_set	unordered_multiset	unordered_map	unordered_multimap
	(destructor)	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap
	operator=	operator=	operator=	operator=	operator=
	assign				
Ітератори	begin	begin	begin	begin	begin

	cbegin	cbegin	cbegin	cbegin	cbegin
	end	end	end	end	end
	cend	cend	cend	cend	cend
	rbegin				
	crbegin				
	rend				
	crend				
Доступ до елементів	at			at	
	operator[]			operator[]	
	data				
	front				
	back				
Ємність	empty	empty	empty	empty	empty
	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size
	resize				
	capacity	bucket_count	bucket_count	bucket_count	bucket_count
	reserve	reserve	reserve	reserve	reserve
	shrink_to_fit				
Модифікатори	clear	clear	clear	clear	clear
	insert	insert	insert	insert	insert
	insert_or_assign			insert_or_assign	
	emplace	emplace	emplace	emplace	emplace
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint
	try_emplace			try_emplace	
	erase	erase	erase	erase	erase
	push_front				
	emplace_front				
	pop_front				
	push_back				
	emplace_back				
	pop_back				
	swap	swap	swap	swap	swap
	merge	merge	merge	merge	merge
	extract	extract	extract	extract	extract
Операції з контейнером	splice				
	remove				
	remove_if				
	reverse				
	unique				
	sort				
Пошук	count	count	count	count	count
	find	find	find	find	find
	contains	contains	contains	contains	contains
	lower_bound				
	upper_bound				
	equal_range	equal_range	equal_range	equal_range	equal_range
Спостерега	key_comp				

чі	value_comp				
	hash_function	hash_function	hash_function	hash_function	hash_function
	key_eq	key_eq	key_eq	key_eq	key_eq
Розподілювач	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator
Контейнер		unordered_set	unordered_multiset	unordered_map	unordered_multimap
Невпорядковані асоціативні контейнери					

		Адаптери контейнерів		
Заголовок		<stack>	<queue>	
Контейнер		stack	queue	priority_queue
	(constructor)	stack	queue	priority_queue
	(destructor)	~stack	~queue	~priority_queue
	operator=	operator=	operator=	operator=
	assign			
Ітератори	begin			
	cbegin			
	end			
	cend			
	rbegin			
	crbegin			
	rend			
	crend			
Доступ до елементів	at			
	operator[]			
	data			
	front		front	top
	back	top	back	
Ємність	empty	empty	empty	empty
	size	size	size	size
	max_size			
	resize			
	capacity			
	reserve			
	shrink_to_fit			
Модифікатори	clear			
	insert			
	insert_or_assign			
	emplace			
	emplace_hint			
	try_emplace			
	erase			
	push_front			
	emplace_front			
	pop_front		pop	pop
	push_back	push	push	push
	emplace_back	emplace	emplace	emplace

	pop_back	pop		
	swap	swap	swap	swap
	merge			
	extract			
Операції з контейнером	splice			
	remove			
	remove_if			
	reverse			
	unique			
	sort			
Пошук	count			
	find			
	contains			
	lower_bound			
	upper_bound			
	equal_range			
Спостерігачі	key_comp			
	value_comp			
	hash_function			
	key_eq			
Розподільвач	get_allocator			
Контейнер		stack	queue	priority_queue
		Адаптери контейнерів		

## Ітератори

Для використання ітераторів слід підключити файл заголовку `<iterator>`.

Бібліотека ітераторів надає визначення ітераторів для п'яти (до C++17) шести (починаючи з C++17) типів, а також властивостей ітераторів, адаптерів і службових функцій.

### Визначення ітератора

Ітератор – це будь-який об'єкт, який вказує на певний елемент у діапазоні елементів (наприклад, масив або контейнер) і має можливість перебирати елементи цього діапазону, використовуючи набір операцій (як мінімум – інкремент ++ та роз'іменування \*).

Найбільш очевидною формою ітератора є вказівник – він може вказувати на елементи в масиві і може здійснювати ітерацію через них за допомогою операції інкременту ++. Але можливі й інші види ітераторів. Кожний тип контейнера (наприклад, список) має певний тип ітератора, призначений для ітерації по його елементах.

Хоча вказівник є формою ітератора, не всі ітератори мають однакову функціональність вказівників. Залежно від властивостей, що підтримуються ітераторами, їх класифікують на п'ять (шість – починаючи з C++17) різних категорій.

*Ітератори STL* забезпечують доступ до елементів контейнера. Всі контейнери надають методи для присвоєння початкового та кінцевого значення ітераторам. Ітератори контейнерів реалізовані як вкладені класи, тому при діях з контейнерами можна використовувати і відповідні ітератори. Наприклад, при роботі з контейнером-вектором

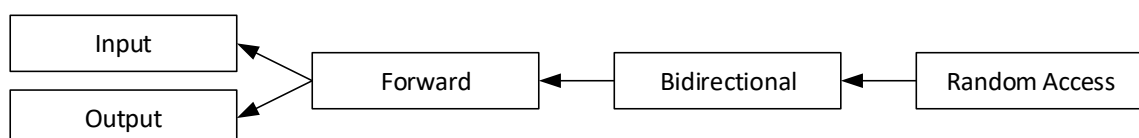
```
vector<int> L;
```

можна оголосити в програмі і відповідний ітератор:

```
vector<int>::iterator il = L.begin();
```

### Категорії ітераторів

Ітератори класифікуються на категорії – залежно від функціональності, яку вони реалізують:



Ітератори введення та виведення (вхідні – **Input**, вихідні – **Output**) є найбільш обмеженими типами ітераторів: вони можуть виконувати послідовні однопрохідні операції введення або виведення.

Прямі (Forward) ітератори мають всю функціональність вхідних ітераторів, а якщо вони не є постійними ітераторами, також функціональність вихідних ітераторів, хоча вони обмежені одним напрямком, в якому здійснюється ітерація через діапазон (вперед). Усі стандартні контейнери підтримують принаймні прямі типи ітераторів.

Двонаправлені (Bidirectional) ітератори схожі на прямі ітератори, але їх також можна перебирати назад.

Ітератори довільного доступу (Random Access) реалізують всю функціональність двонаправлених ітераторів, а також мають можливість не послідовно отримувати доступ до діапазонів: до елементів можна отримати прямий доступ, застосовуючи значення зміщення до ітератора, не перебираючи всі елементи між ними. Ці ітератори мають схожу функціональність із стандартними вказівниками (вказівники є ітераторами цієї категорії).

Властивості кожної категорії ітераторів:

Категорія				Властивості	Вірні вирази	
Всі категорії				Конструктор копіювання, присвоєння (копіювання), деструктор (знищення)	X b(a); b = a;	
				Може бути інкрементований	++a a++	
Random Access	Bidirectional	Forward	Input	Підтримка порівняння на збіг / відмінність	a == b a != b	
				Може бути роз-іменований як <i>rvalue</i>	*a a->m	
		Output	Може бути роз-іменований як <i>lvalue</i> (лише для змінних типів ітераторів)	*a = t *a++ = t		
			Конструктор за умовчанням	X a; X()		
				Багато прохідність: ані роз-іменування, ані інкремент не впливають на можливість роз-іменування	{ b=a; *a++; *b; }	
					Може бути декрементований	-- a a-- *a--
					Підтримка арифметичних операцій + та -	a + n n + a a - n a - b
					Підтримка порівнянь на нерівність (<, >, <= та >=) між ітераторами	a < b a > b a <= b a >= b
					Підтримка арифметичних операцій присвоєння += та -=	a += n a -= n
					Підтримка операції індексування [] (роз-іменування зі зміщенням)	a[n]

Тут X – тип ітератора, a і b – об'єкти цього типу ітератора, t – об'єкт типу, вказаний типом ітератора, a n – ціле число.



Крім п'яти вище зазначених до версії C++17, є шостий – починаючи з C++17 вид ітераторів:

- LegacyInputIterator,
- LegacyOutputIterator,
- LegacyForwardIterator,
- LegacyBidirectionalIterator,
- LegacyRandomAccessIterator і
- LegacyContiguousIterator (починаючи з C++17).

Замість того, щоб визначатися конкретними типами, кожна категорія ітераторів визначається операціями, які можуть бути виконані з нею. Це означає, що будь-який тип, що підтримує необхідні операції, може використовуватися в якості ітератора – наприклад, вказівник підтримує всі операції, необхідні для LegacyRandomAccessIterator, тому вказівник можна використовувати де завгодно, де очікується LegacyRandomAccessIterator.

Всі категорії ітераторів (крім LegacyOutputIterator) можуть бути організовані в ієрархію, де більш потужні категорії ітераторів (наприклад, LegacyRandomAccessIterator) підтримують операції менш потужних категорій (наприклад, LegacyInputIterator). Якщо ітератор потрапляє в одну з цих категорій і також відповідає вимогам LegacyOutputIterator, то він називається змінним ітератором і підтримує введення і виведення. Незмінні ітератори називаються константними ітераторами.

Категорія ітератора					Операції
<u>LegacyContiguousIterator</u>	<u>LegacyRandomAccessIterator</u>	<u>LegacyBidirectionalIterator</u>	<u>LegacyForwardIterator</u>	<u>LegacyInputIterator</u>	зчитування, інкремент (без кількох проходів)
					інкремент (з кількома проходимаи)
					декремент
					довільний доступ
					неперервне сховище
Ітератори, які належать одній із вищевказаних категорій і відповідають вимогам <u>LegacyOutputIterator</u> , називаються змінюваними ітераторами.					
<u>LegacyOutputIterator</u>					запис, інкремент (без кількох проходів)

**Зауваження:** категорія `LegacyContiguousIterator` була вказана в C++17 лише формально, але ітератори `std::vector`, `std::basic_string`, `std::array` та `std::valarray`, а також вказівники на масиви C часто розглядаються як окрема категорія в коді до C++17.

Таким чином, в STL визначено наступні категорії ітераторів – в порядку збільшення «можливостей»:

- вхідний (Input) – зчитування елементів контейнера в прямому порядку;
- вихідний (Output) – запис елементів контейнера в прямому порядку;
- прямий (Forward) – зчитування та запис елементів контейнера в прямому порядку;
- двосторонній (Bidirectional) – зчитування та запис елементів контейнера в прямому та зворотному порядках;
- довільного доступу (Random Access) – довільний доступ для зчитування та запису елементів контейнера.

На практиці для будь-якого контейнера гарантується двосторонній ітератор. Крім того, для всіх контейнерів, крім списку, забезпечується ітератор довільного доступу, який реалізує повну арифметику вбудованих вказівників.

Ітератори можуть бути константними та неконстантними. Константні ітератори не використовуються для зміни елементів контейнера.

Ітератори можуть бути дійсними (коли ітератору відповідає елемент контейнера) і недійсними (при відсутності відповідного елемента контейнера) – мова не йде про тип `float` чи `double`! Ітератор може бути недійсним із-за трьох причин:

- ітератор не був ініціалізований;
- ітератор дорівнює `end()`;
- контейнер, з яким пов'язаний ітератор, змінив розміри чи взагалі був знищений.

Останнє часто стає джерелом помилок в програмах, бо програмісти забувають, що **при видаленні чи вставці елемента ітератори стають недійсними.**

Для всіх категорій ітераторів визначені наступні операції (і та j – ітератори):

- `i++` – зміщення вперед (повертає стару позицію);
- `++i` – зміщення вперед (повертає нову позицію);
- `i = j` – присвоєння ітераторів;
- `i == j` – порівняння ітераторів на збіг;
- `i != j` – порівняння ітераторів на відмінність;

- `*i` – звертання до елемента;
- `i->member` – звертання до поля/метода елемента; еквівалентно `(*i).member`.

Інші операції, визначені для різних категорій ітераторів, показані в наступній таблиці.

### Операції з ітераторами різних категорій

Категорія ітератора	Операції	Контейнери
Вхідний	<code>x = *i</code> – зчитування елемента	всі
Вихідний	<code>*i = x</code> – запис елемента	всі
Прямий	<code>x = *i, *i = x</code>	всі
Двосторонній	<code>x = *i, *i = x, --i, i--</code>	всі
Довільного доступу	<code>x = *i, *i = x, --i, i--, i + n, i - n, i += n, i -= n, i &lt; j, i &gt; j, i &lt;= j, i &gt;= j</code>	всі, крім <code>list</code>

## Адаптери ітераторів

Адаптери ітераторів – це: зворотні ітератори, ітератори вставки, потокові ітератори.

Методи `rbegin()` та `rend()`, які реалізовані для всіх контейнерів, повертають початкове та кінцеве значення *зворотного ітератора*. Використання зворотного ітератора нічим не відрізняється від використання прямого ітератора, лише операція `++` визначена так, що переміщення по контейнеру здійснюється від останнього елемента до першого. Зворотні ітератори дозволяють відсортувати контейнер у зворотному порядку.

Для використання *ітераторів вставки* та *потоків ітераторів* слід підключити файл заголовку

```
#include <iterator>
```

Ітератори вставки і потокові ітератори зазвичай використовуються в поєднанні з алгоритмами модифікації. Алгоритми модифікації працюють в режимі заміщення: результуючий контейнер на момент виклику алгоритму має існувати і у ньому має бути достатньо елементів. Попередні значення елементів замінюються новими.

*Ітератори вставки* дозволяють вставляти в існуючий порожній контейнер нові елементи. STL надає три види ітераторів вставки:

- `back_insert_iterator` – вставка в кінець контейнера;
- `front_insert_iterator` – вставка в початок контейнера;
- `insert_iterator` – вставка перед заданим елементом.

Ітератор вставки в кінець (кінцевий ітератор) працює з будь-яким послідовним контейнером. Ітератор вставки в початок (початковий ітератор) неможна використовувати з

вектором, бо вектор не забезпечує вставку елементів в початок – для нього не визначено метод `push_front()`. Третій вид ітераторів підходить для будь-якого контейнера.

Безпосередньо звертатися до ітераторів вставки не дуже зручно, тому в бібліотеці реалізовані функції-обгортки у вигляді шаблонів. Зокрема, кінцевий ітератор для контейнера створюється функцією `back_inserter()`:

```
template<class Container>
back_insert_iterator<Container> back_inserter(Container& x);
```

Аналогічно виглядає і функція `front_inserter()`:

```
template<class Container>
front_insert_iterator<Container> front_inserter(Container& x);
```

Обидві функції отримують в якості аргументу контейнер, в який буде виконуватися вставка, і повертають відповідний ітератор вставки.

Нарешті, функція `inserter()` забезпечує роботу з довільним ітератором вставки. В якості аргументів, крім контейнера, передається ще ітератор, який визначає позицію вставки (елемент, перед яким буде виконана вставка):

```
template<class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i);
```

*Потокові ітератори* – це ітератори у вигляді адаптерів, які сприймають потік даних за джерело чи приймач даних. Потоковий ітератор вводу зчитує елементи із потоку даних, а потоковий ітератор виводу – записує дані у потік.

*Потоковий ітератор виводу* створюється одним із конструкторів:

- `ostream_iterator<T>(stream, delim)` – створює ітератор, пов'язаний з потоком виводу `stream`, в якому значення при виводі відокремлюються літерним рядком `delim` типу `const char *`;
- `ostream_iterator<T>(stream)` – створює ітератор, пов'язаний з потоком виводу `stream`, в якому значення при виводі не відокремлюються.

Виведення в потік виконується операцією присвоєння, а переміщення – операцією `++`, як і переміщення ітераторів вставки. Можна використовувати ітератор виводу безпосередньо, проте зазвичай ітератор виводу використовується одночасно з алгоритмами:

```
#include <iterator>
#include <algorithm>
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    int a[100] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```

// вивід 10 елементів контейнера на екран
copy(a, a + 10, ostream_iterator<int>(cout, " "));
cout << endl;
// "1, 2, 3, 4, 5, 6, 7, 8, 9, 10, "

// вивід трьох значень "5; "
fill_n(ostream_iterator<int>(cout, "; "), 3, 5);
// "5; 5; 5; "

// вивід у файл integers.out 10 елементів контейнера по одному в рядку
ofstream outfile("integers.out");
copy(a, a + 10, ostream_iterator<int>(outfile, "\n"));

return 0;
}

```

Потоковий ітератор вводу дозволяє алгоритмам зчитувати дані не з контейнера, а з потоку введення. Для отримання даних з потоку слід мати два ітератори: *потоковий ітератор вводу* та *ітератор кінця потоку*. Ці ітератори створюються конструкторами:

- `istream_iterator<T>(stream)` – створює ітератор, пов'язаний з потоком вводу `stream`;
- `istream_iterator<T>()` – створює ітератор кінця потоку.

Якщо спроба зчитування – невдала (наприклад, виникає ситуація end-of-file), то потоковий ітератор вводу перетворюється в ітератор кінця потоку.

Ітератори вводу можна визначати окремими змінними, а можна використовувати анонімні ітератори безпосередньо як аргументи при виклику алгоритмів:

```

#include <vector>
#include <iterator>
#include <algorithm>
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    vector<int> v(10);
    cout << "enter values:" << endl;

    // оголошення ітераторів
    istream_iterator<int> Read(cin);           // потоковий ітератор вводу
    istream_iterator<int> end;                 // ітератор кінця потоку
    copy(Read, end, inserter(v, v.begin())); // вставка у вектор

    // без оголошення ітераторів
    copy(istream_iterator<int>(cin),           // потоковий ітератор вводу
        istream_iterator<int>(),              // ітератор кінця потоку
        inserter(v, v.begin()));              // вставка у вектор

    // слід натиснути CTRL+Z та ENTER для завершення вводу

    // зчитування рядків з файлу
    ifstream infile("integers.out");           // вхідний файл
    istream_iterator<int> is(infile);          // ітератор вхідного потоку
                                              // зв'язали з файлом
    istream_iterator<int> fend;                // ітератор кінця потоку

```

```

vector<int> b;
copy(is, fend, inserter(b, b.begin())); // зчитування з файлу у вектор

copy(b.begin(), b.end(), ostream_iterator<int>(cout, ", "));
cout << endl;
// або
for_each(b.begin(), b.end(), [](const int& n) { cout << n << ", "; });
cout << endl;

return 0;
}

```

Останній цикл `for_each` пояснює наступний приклад, який використовує лямбда-функцію для збільшення всіх елементів вектора, а потім використовує перевантажену операцію `()` у функторі для обчислення їх суми. Зауваження: для обчислення суми рекомендується використовувати алгоритм `std::accumulate`.

```

#include <vector>
#include <algorithm>
#include <iostream>

struct Sum
{
    void operator()(int n) { sum += n; } // функтор - накопичує суму
    int sum{ 0 }; // в це поле; { 0 } - ініціалізатор
};

int main()
{
    std::vector<int> nums{ 3, 4, 2, 8, 15, 267 }; // створили і заповнили вектор

    // auto - тип виводить компілятор
    auto print = [](const int& n) { std::cout << " " << n; }; // лямбда-функція

    std::cout << "before:";
    std::for_each(nums.cbegin(), nums.cend(), print);
    std::cout << '\n';

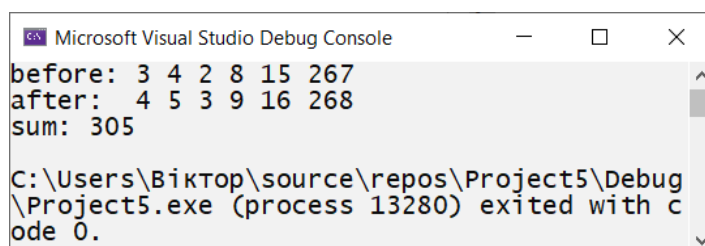
    // 3-й параметр - лямбда-функція
    std::for_each(nums.begin(), nums.end(), [](int& n) { n++; });

    // виклик функтора Sum::operator() для кожного елемента контейнера
    Sum s = std::for_each(nums.begin(), nums.end(), Sum());

    std::cout << "after: ";
    std::for_each(nums.cbegin(), nums.cend(), print);
    std::cout << '\n';
    std::cout << "sum: " << s.sum << '\n';
}

```

Результат виконання:



```

Microsoft Visual Studio Debug Console
before: 3 4 2 8 15 267
after: 4 5 3 9 16 268
sum: 305

C:\Users\Віктор\source\repos\Project5\Debug\Project5.exe (process 13280) exited with code 0.

```

## Допоміжні функції для ітераторів

Для більшої зручності STL надає функції для роботи з ітераторами:

```
// перемістити ітератор
template<class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);

// обчислити відстань між ітераторами
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Щоб використовувати ці функції, слід підключити файл

```
#include <iterator>
```

Функція `advance()` дозволяє виконати операції `i += n` та `i -= n` з ітератором будь-якої категорії, а не лише з ітератором довільного доступу.

Функція `distance()` обчислює відстань між двома ітераторами, які посилаються на елементи одного контейнера. Відстань – це значення типу `difference_type`, яке є різницею `last - first`, і фактично визначає кількість елементів у цьому діапазоні.

## Концепти ітераторів C++20

В мові C++20 представлена нова система ітераторів, заснована на концептах, яка відрізняється від ітераторів C++17. Хоча основна схема залишається схожою, вимоги до окремих категорій ітераторів дещо відрізняються.

Визначені в просторі імен <code>std</code>	
<code>indirectly_readable</code> (C++20)	вказує, що тип доступний для читування за допомогою операції *
<code>indirectly_writable</code> (C++20)	вказує, що значення може бути записано в об'єкт, на який вказує посилання ітератора
<code>weakly_incrementable</code> (C++20)	вказує, що тип <code>semiregular</code> може інкрементувати за допомогою операцій пре-інкременту і пост-інкременту
<code>incrementable</code> (C++20)	вказує, що операція інкремента для типу <code>weakly_incrementable</code> зберігає рівність і що цей тип <code>equality_comparable</code>
<code>input_or_output_iterator</code> (C++20)	вказує, що об'єкти типу можуть бути інкрементовані та розіменовані
<code>sentinel_for</code> (C++20)	вказує, що тип є обмежувачем для типу <code>input_or_output_iterator</code>
<code>sized_sentinel_for</code> (C++20)	вказує, що операція - може бути застосований до ітератора і обмежувача, щоб обчислити їх різницю за постійний час
<code>input_iterator</code> (C++20)	вказує, що тип є ітератором вводу, тобто значення, на які він посилається, можуть бути прочитані, і він може бути як пре-інкрементованим, так і пост-інкрементованим
<code>output_iterator</code> (C++20)	вказує, що тип є ітератором виводу для цього типу значення, тобто в нього можуть бути записані значення цього типу, і він може бути як пре-інкрементованим, так і пост-інкрементованим

<code>forward_iterator</code> (C++20)	вказує, що <code>input_iterator</code> є прямим ітератором, що підтримує порівняння на рівність і багатопрохідність
<code>bidirectional_iterator</code> (C++20)	вказує, що <code>forward_iterator</code> є двонаправленим ітератором, що підтримує рух назад
<code>random_access_iterator</code> (C++20)	вказує, що <code>bidirectional_iterator</code> це ітератор з довільним доступом, що підтримує просування за постійний час і індексацію
<code>contiguous_iterator</code> (C++20)	вказує, що <code>random_access_iterator</code> є безперервним ітератором, що посиляється на суміжні елементи в пам'яті.

## Функції

### Операції з ітераторами

<code>advance</code>	Переміщення ітератора
<code>distance</code>	Повернути відстань між ітераторами
<code>begin</code>	Ітератор до початку
<code>end</code>	Ітератор до кінця
<code>cbegin</code>	Константний ітератор до початку
<code>cend</code>	Константний ітератор до кінця
<code>rbegin</code>	Зворотний ітератор до початку
<code>rend</code>	Зворотний ітератор до кінця
<code>prev</code>	Отримати ітератор до попереднього елемента
<code>next</code>	Отримати ітератор до наступного елемента

### `advance`

```
template <class InputIterator, class Distance>
void advance(InputIterator& it, Distance n);
```

Переміщує ітератор на `n` позицій (елементів).

Якщо це – ітератор довільного доступу, функція використовує лише один раз операцію `operator+` або `operator-`. В іншому випадку функція використовує операції інкременту або декременту (`operator++` або `operator--`), поки ітератор не буде переміщено на `n` елементів.

Приклад:

```
// advance example
#include <iostream>          // std::cout
#include <iterator>          // std::advance
#include <list>              // std::list

int main()
{
    std::list<int> mylist;
    for (int i = 0; i < 10; i++) mylist.push_back(i * 10);

    std::list<int>::iterator it = mylist.begin();

    std::advance(it, 5);
```



```

std::cout << "The sixth element in mylist is: " << *it << '\n';
// The sixth element in mylist is: 50

return 0;
}

```

## distance

```

template<class InputIterator>
typename std::iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);

```

Повертає відстань між ітераторами

Обчислює кількість елементів між first та last.

Приклад:

```

// distance example
#include <iostream>      // std::cout
#include <iterator>      // std::distance
#include <list>          // std::list

int main()
{
    std::list<int> mylist;
    for (int i = 0; i < 10; i++) mylist.push_back(i * 10);

    std::list<int>::iterator first = mylist.begin();
    std::list<int>::iterator last = mylist.end();

    std::cout << "The distance is: " << std::distance(first, last) << '\n';
    // The distance is: 10

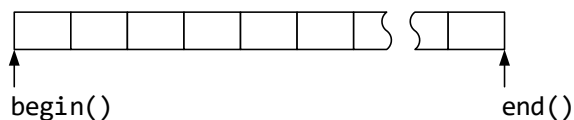
    return 0;
}

```

## begin / end

begin() – ітератор початку. Повертає ітератор, що вказує на перший елемент послідовності.

end() – ітератор кінця. Повертає ітератор, що вказує на позицію після останнього елемента послідовності.



Приклад:

```

// std::begin / std::end example
#include <iostream>      // std::cout
#include <vector>         // std::vector, std::begin, std::end

int main()
{
    int foo[] = { 10, 20, 30, 40, 50 };
    std::vector<int> bar;
}

```

```

// iterate foo: inserting into bar
for (auto it = std::begin(foo); it != std::end(foo); ++it)
    bar.push_back(*it);

// iterate bar: print contents:
std::cout << "bar contains:";
for (auto it = std::begin(bar); it != std::end(bar); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// bar contains: 10 20 30 40 50

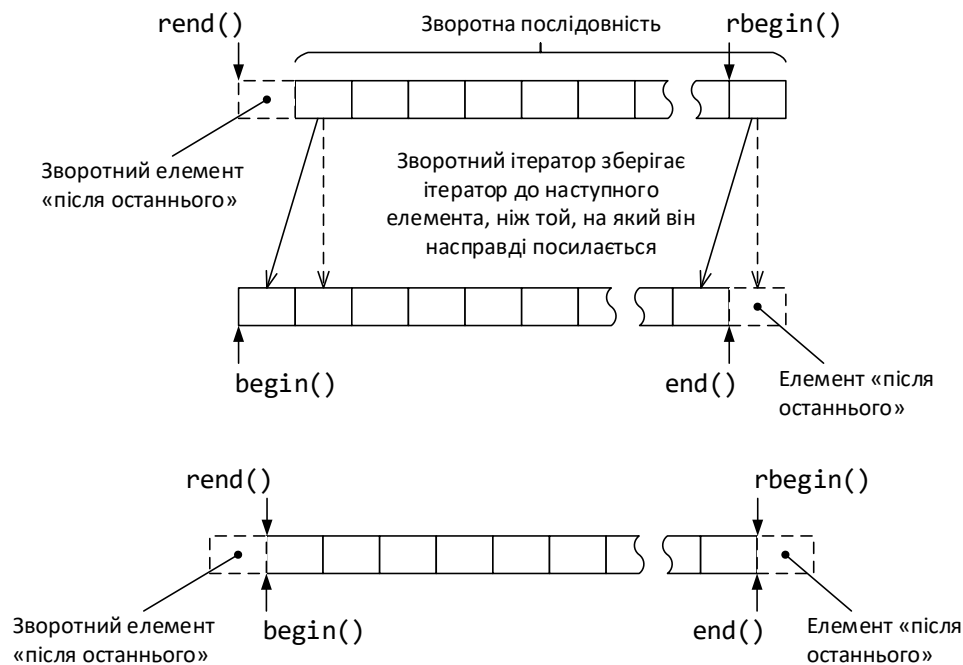
return 0;
}

```

## rbegin / rend

`rbegin()` – повертає ітератор до зворотного початку заданого діапазону.

`rend()` – повертає ітератор до зворотного кінця заданого діапазону.



## Приклад:

```

// std::rbegin / std::rend example
#include <iostream>
#include <vector>
#include <iterator>

int main()
{
    std::vector<int> v = { 3, 1, 4 };
    auto vi = std::rbegin(v); // the type of 'vi' is std::vector<int>::reverse_iterator
    std::cout << "*vi = " << *vi << '\n';

    *std::rbegin(v) = 42;      // OK: after assignment v[2] == 42
    // *std::crbegin(v) = 13;   // error: the location is read-only

    int a[] = { -5, 10, 15 };
    auto ai = std::rbegin(a); // the type of 'ai' is std::reverse_iterator<int*>
    std::cout << "*ai = " << *ai << '\n';
}

```

```

    auto il = { 3, 1, 4 };
    // the type of 'it' below is std::reverse_iterator<int const*>:
    for (auto it = std::rbegin(il); it != std::rend(il); ++it)
        std::cout << *it << ' ';
    // *vi = 4
    // *ai = 15
    // 4 1 3

    return 0;
}

```

## prev

```

template <class BidirectionalIterator>
    BidirectionalIterator prev(BidirectionalIterator it,
        typename std::iterator_traits<BidirectionalIterator>::difference_type n = 1);

```

Ітератор до попереднього елемента

Повертає ітератор, що вказує на елемент, на який вказував би it, якщо перемістити позицію на -n елементів (назад, до початку послідовності).

Приклад:

```

// prev example
#include <iostream>      // std::cout
#include <iterator>      // std::next
#include <list>          // std::list
#include <algorithm>     // std::for_each

int main()
{
    std::list<int> mylist;
    for (int i = 0; i < 10; i++) mylist.push_back(i * 10);

    std::cout << "The last element is " << *std::prev(mylist.end()) << '\n';
    // The last element is 90

    return 0;
}

```

## next

```

template <class ForwardIterator>
    ForwardIterator next(ForwardIterator it,
        typename std::iterator_traits<ForwardIterator>::difference_type n = 1);

```

Ітератор до наступного елемента

Повертає ітератор, що вказує на елемент, на який вказував би it, якщо перемістити позицію на n елементів (вперед, до кінця послідовності).

Приклад:

```

// next example
#include <iostream>      // std::cout
#include <iterator>      // std::next
#include <list>          // std::list
#include <algorithm>     // std::for_each

int main()
{

```

```

std::list<int> mylist;
for (int i = 0; i < 10; i++) mylist.push_back(i * 10);

std::cout << "mylist:";
std::for_each(mylist.begin(),
    std::next(mylist.begin(), 5),
    [](int x) {std::cout << ' ' << x; });

std::cout << '\n';
// mylist: 0 10 20 30 40

return 0;
}

```

## Генератори ітераторів

back_inserter	Створює ітератор вставки в кінці контейнера
front_inserter	Створює ітератор вставки на початку контейнера
inserter	Створює ітератор вставки
make_move_iterator	Створює ітератор переміщення

### back\_inserter

```

template <class Container>
std::back_inserter<Container> back_inserter(Container& x);

```

Створює ітератор вставки в кінець, який вставляє нові елементи в кінці контейнера x.

Ітератор вставки в кінець — це спеціальний тип ітератора виводу, призначений для того, щоб алгоритми, які зазвичай перезаписують елементи (наприклад, копіювання copy), замість цього автоматично вставляли нові елементи в кінці контейнера.

Контейнер x мусить мати метод push\_back (наприклад, як стандартні контейнери vector, deque та list).

Приклад:

```

// back_inserter example
#include <iostream>      // std::cout
#include <iterator>      // std::back_inserter
#include <vector>        // std::vector
#include <algorithm>     // std::copy

int main()
{
    std::vector<int> foo, bar;
    for (int i = 1; i <= 5; i++)
    {
        foo.push_back(i); bar.push_back(i * 10);
    }

    std::copy(bar.begin(), bar.end(), back_inserter(foo));

    std::cout << "foo contains:";
    for (std::vector<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
}

```

```

        // foo contains: 1 2 3 4 5 10 20 30 40 50
    return 0;
}

```

## front\_inserter

```

template <class Container>
    std::front_insert_iterator<Container> front_inserter(Container& x);

```

Створює ітератор вставки на початок, який вставляє нові елементи на початку контейнера *x*.

Ітератор вставки на початок – це спеціальний тип ітератора виводу, призначений для того, щоб алгоритми, які зазвичай перезаписують елементи (наприклад, копіювання *copy*), замість цього автоматично вставляли нові елементи на початку контейнера.

Контейнер *x* мусить мати метод *push\_front* (наприклад, як стандартні контейнери *deque* та *list*).

Приклад:

```

// front_inserter example
#include <iostream>      // std::cout
#include <iterator>      // std::front_inserter
#include <deque>         // std::deque
#include <algorithm>     // std::copy

int main()
{
    std::deque<int> foo, bar;
    for (int i = 1; i <= 5; i++)
    {
        foo.push_back(i); bar.push_back(i * 10);
    }

    std::copy(bar.begin(), bar.end(), std::front_inserter(foo));

    std::cout << "foo contains:";
    for (std::deque<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // 50 40 30 20 10 1 2 3 4 5

    return 0;
}

```

## inserter

```

template <class Container, class Iterator>
    std::insert_iterator<Container> inserter(Container& x, Iterator it);

```

Створює ітератор вставки, який послідовно вставляє нові елементи в контейнер *x*, починаючи з вказаної ітератором *it* позиції.

Ітератор вставки – це спеціальний тип ітератора виводу, призначений для того, щоб алгоритми, які зазвичай перезаписують елементи (наприклад, копіювання *copy*), замість цього автоматично вставляли нові елементи в певну позицію в контейнері.

Контейнер `x` мусить мати метод `insert` (як більшість стандартних контейнерів).

Приклад:

```
// inserter example
#include <iostream>      // std::cout
#include <iterator>      // std::front_inserter
#include <list>          // std::list
#include <algorithm>     // std::copy

int main()
{
    std::list<int> foo, bar;
    for (int i = 1; i <= 5; i++)
    {
        foo.push_back(i); bar.push_back(i * 10);
    }

    std::list<int>::iterator it = foo.begin();
    advance(it, 3);

    std::copy(bar.begin(), bar.end(), std::inserter(foo, it));

    std::cout << "foo contains:";
    for (std::list<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // 1 2 3 10 20 30 40 50 4 5

    return 0;
}
```

### **make\_move\_iterator**

```
template <class Iterator>
std::move_iterator<Iterator> make_move_iterator(const Iterator& it);
```

Створює об'єкт класу `move_iterator` з ітератора `it`.

`make_move_iterator` – це зручний шаблон функції, який створює `std::move_iterator` для заданого ітератора. Результат функції: `std::move_iterator`, який можна використовувати для переміщення елементів, доступ до яких здійснюється через ітератор `it`.

Приклад 1:

```
// make_move_iterator example
#include <iostream>      // std::cout
#include <iterator>      // std::make_move_iterator
#include <vector>         // std::vector
#include <string>         // std::string
#include <algorithm>     // std::copy

int main()
{
    std::vector<std::string> foo(3);
    std::vector<std::string> bar{ "one", "two", "three" };

    std::copy(make_move_iterator(bar.begin()),
              make_move_iterator(bar.end()),
              foo.begin());

    // bar now contains unspecified values; clear it:
}
```

```

bar.clear();

std::cout << "foo:";
for (std::string& x : foo) std::cout << ' ' << x;
std::cout << '\n';
// foo: one two three

return 0;
}

```

## Приклад 2:

```

// make_move_iterator example
#include <iostream>
#include <iomanip>
#include <list>
#include <vector>
#include <string>
#include <iterator>

auto print = [](auto const& rem, auto const& seq)
{
    std::cout << rem;
    for (auto const& str : seq)
        std::cout << std::quoted(str) << ' ';
    std::cout << '\n';
};

int main()
{
    std::list<std::string> s{ "one", "two", "three" };

    std::vector<std::string> v1(s.begin(), s.end());           // copy

    std::vector<std::string> v2(std::make_move_iterator(s.begin()),
                                std::make_move_iterator(s.end())); // move

    print("v1 now holds: ", v1);
    print("v2 now holds: ", v2);
    print("original list now holds: ", s);
    // v1 now holds: "one" "two" "three"
    // v2 now holds: "one" "two" "three"
    // original list now holds: "" "" ""

    return 0;
}

```

## Класи

iterator	Базовий клас ітератора
iterator_traits	Характеристики ітератора

## iterator

```

template <class Category,           // iterator::iterator_category
         class T,                  // iterator::value_type
         class Distance = ptrdiff_t, // iterator::difference_type
         class Pointer = T*,        // iterator::pointer
         class Reference = T&       // iterator::reference
>
struct iterator

```

```
{
    typedef T          value_type;
    typedef Distance   difference_type;
    typedef Pointer     pointer;
    typedef Reference   reference;
    typedef Category    iterator_category;
};
```

`std::iterator` – шаблон базового класу, який надається для спрощення визначень необхідних типів для ітераторів. Його можна використовувати для виведення з нього класів ітераторів. Це не клас ітераторів і він не забезпечує жодної функціональності, яку, як очікується, має ітератор.

Цей шаблон базового класу надає лише деякі типи елементів класу, які насправді не повинні бути присутніми в будь-якому типі ітератора (типи ітераторів не мають певних вимог до елементів класу).

Приклад:

```
// std::iterator example
#include <iostream>      // std::cout
#include <iterator>      // std::iterator, std::input_iterator_tag

class MyIterator : public std::iterator<std::input_iterator_tag, int>
{
    int* p;
public:
    MyIterator(int* x) : p(x) {}
    MyIterator(const MyIterator& mit) : p(mit.p) {}
    MyIterator& operator++() { ++p; return *this; }
    MyIterator operator++(int) { MyIterator tmp(*this); operator++(); return tmp; }
    bool operator==(const MyIterator& rhs) const { return p == rhs.p; }
    bool operator!=(const MyIterator& rhs) const { return p != rhs.p; }
    int& operator*() { return *p; }
};

int main()
{
    int numbers[] = { 10,20,30,40,50 };
    MyIterator from(numbers);
    MyIterator until(numbers + 5);
    for (MyIterator it = from; it != until; it++)
        std::cout << *it << ' ';
    std::cout << '\n';
    // 10 20 30 40 50

    return 0;
}
```

## iterator\_traits

Клас ознак, які визначають властивості ітераторів.

Стандартні алгоритми визначають певні властивості переданих їм ітераторів та діапазон, який вони представляють, за допомогою елементів відповідного екземпляра `iterator_traits`.

Приклад:



```

// iterator_traits example
#include <iostream>      // std::cout
#include <iterator>      // std::iterator_traits
#include <typeinfo>      // typeid

int main()
{
    typedef std::iterator_traits<int*> traits;
    if (typeid(traits::iterator_category) == typeid(std::random_access_iterator_tag))
        std::cout << "int* is a random-access iterator";
    // int* is a random-access iterator

    return 0;
}

```

## Визначені в STL ітератори

reverse_iterator	Зворотний ітератор
move_iterator	Ітератор переміщення
back_insert_iterator	Ітератор вставки в кінець
front_insert_iterator	Ітератор вставки на початок
insert_iterator	Ітератор вставки
istream_iterator	Ітератор зчитування
ostream_iterator	Ітератор запису
istreambuf_iterator	Ітератор буфера вхідного потоку
ostreambuf_iterator	Ітератор буфера вихідного потоку

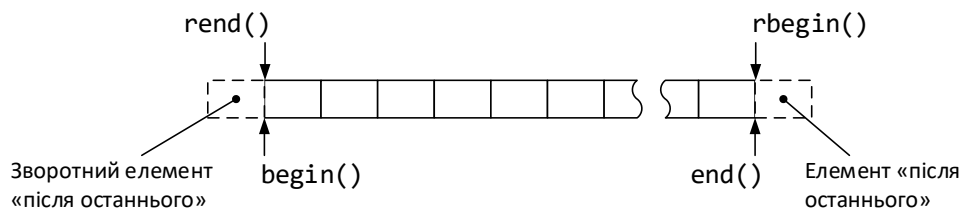
## reverse\_iterator

```
template <class Iterator> class reverse_iterator;
```

Цей клас змінює напрямок, у якому двонаправлений ітератор або ітератор довільного доступу перебирає діапазон.

Копія оригінального ітератора (базового ітератора) зберігається внутрішньо і використовується для відображення операцій, виконаних з reverse\_iterator: щоразу, коли збільшується reverse\_iterator, його базовий ітератор зменшується, і навпаки. Копію базового ітератора з поточним станом можна отримати в будь-який час.

Зауваження: реверсована версія ітератора вказує не на той самий елемент у діапазоні, а на той, що йому передує. Це зроблено для того, щоб організувати елемент після-останнього в діапазоні: ітератор, який вказує на елемент після-останнього у діапазоні, при реверсуванні стане вказувати на останній елемент діапазону (а не на елемент після-останнього) – це буде перший елемент зворотного діапазону. І якщо ітератор до першого елемента в діапазоні реверсується, реверсований ітератор стане вказувати на елемент перед першим елементом (це буде елемент після-останнього в реверсованому діапазоні).



Приклад:

```
// reverse_iterator example
#include <iostream>          // std::cout
#include <string>             // std::string
#include <iterator>          // std::reverse_iterator

int main()
{
    std::string s = "Hello, world";
    std::reverse_iterator<std::string::iterator> r = s.rbegin();

    r[7] = '0'; // replaces 'o' with '0'
    r += 7;     // iterator now points at '0'

    std::string rev(r, s.rend());
    std::cout << rev << '\n';
    // OlleH

    return 0;
}
```

## move\_iterator

```
template <class Iterator> class move_iterator;
```

Цей клас адаптує ітератор таким чином, щоб при його розмінуванні він створював посилання *rvalue* (як при застосуванні `std::move`). Всі інші операції виконуються як у звичайному ітераторі.

Приклад:

```
// move_iterator example
#include <iostream>          // std::cout
#include <iterator>          // std::move_iterator
#include <vector>            // std::vector
#include <string>            // std::string
#include <algorithm>         // std::copy

int main()
{
    std::vector<std::string> foo(3);
    std::vector<std::string> bar{ "one", "two", "three" };

    typedef std::vector<std::string>::iterator Iter;

    std::copy(std::move_iterator<Iter>(bar.begin()),
              std::move_iterator<Iter>(bar.end()),
              foo.begin());

    // bar now contains unspecified values; clear it:
    bar.clear();
}
```

```

std::cout << "foo:";
for (std::string& x : foo) std::cout << ' ' << x;
std::cout << '\n';
// foo: one two three

return 0;
}

```

## back\_insert\_iterator

```
template <class Container> class back_insert_iterator;
```

Ітератори вставки в кінець — це спеціальні ітератори виведення, призначені для того, щоб алгоритми, які зазвичай перезаписують елементи (наприклад, копіювання сору), замість цього вставляли нові елементи в кінець контейнера.

Контейнер мусить мати метод `push_back` (наприклад, як стандартні контейнери `vector`, `deque` і `list`).

Приклад:

```

// back_insert_iterator example
#include <iostream>      // std::cout
#include <iterator>      // std::back_insert_iterator
#include <vector>        // std::vector
#include <algorithm>     // std::copy

int main()
{
    std::vector<int> foo, bar;
    for (int i = 1; i <= 5; i++)
    {
        foo.push_back(i); bar.push_back(i * 10);
    }

    std::back_insert_iterator< std::vector<int> > back_it(foo);

    std::copy(bar.begin(), bar.end(), back_it);

    std::cout << "foo:";
    for (std::vector<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // foo: 1 2 3 4 5 10 20 30 40 50

    return 0;
}

```

## front\_insert\_iterator

```
template <class Container> class front_insert_iterator;
```

Ітератори вставки на початку — це спеціальні ітератори виводу, призначені для того, щоб алгоритми, які зазвичай перезаписують елементи (наприклад, копіювання сору), замість цього вставляли нові елементи на початку контейнера.

Контейнер мусить мати метод `push_front` (наприклад, як стандартні контейнери `deque` та `list`).

Приклад:

```
// front_insert_iterator example
#include <iostream>      // std::cout
#include <iterator>      // std::front_insert_iterator
#include <deque>         // std::deque
#include <algorithm>     // std::copy

int main()
{
    std::deque<int> foo, bar;
    for (int i = 1; i <= 5; i++)
    {
        foo.push_back(i); bar.push_back(i * 10);
    }

    std::front_insert_iterator< std::deque<int> > front_it(foo);

    std::copy(bar.begin(), bar.end(), front_it);

    std::cout << "foo:";
    for (std::deque<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // 50 40 30 20 10 1 2 3 4 5

    return 0;
}
```

## insert\_iterator

Ітератори вставки – це спеціальні ітератори виводу, призначені для того, щоб алгоритми, які зазвичай перезаписують елементи (наприклад, копіювання `copy`), замість цього вставляли нові елементи у певну позицію в контейнері.

Контейнер мусить мати метод вставки (наприклад, як більшість стандартних контейнерів).

Приклад:

```
// insert_iterator example
#include <iostream>      // std::cout
#include <iterator>      // std::insert_iterator
#include <list>          // std::list
#include <algorithm>     // std::copy

int main()
{
    std::list<int> foo, bar;
    for (int i = 1; i <= 5; i++)
    {
        foo.push_back(i); bar.push_back(i * 10);
    }

    std::list<int>::iterator it = foo.begin();
    advance(it, 3);

    std::insert_iterator< std::list<int> > insert_it(foo, it);

    std::copy(bar.begin(), bar.end(), insert_it);
}
```

```

std::cout << "foo: ";
for (std::list<int>::iterator it = foo.begin(); it != foo.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// 1 2 3 10 20 30 40 50 4 5

return 0;
}

```

## istream\_iterator

```

template <class T, class charT = char, class traits = std::char_traits<charT>,
         class Distance = ptrdiff_t>
class istream_iterator;

```

Ітератори вхідного потоку – це вхідні ітератори, які зчитують послідовні елементи з вхідного потоку (такого, як `cin`). Вони будуються на основі об'єкта класу `basic_istream`. Коли для ітератора використовується операція `operator++`, то він витягує елемент з потоку (за допомогою операції `operator>>`).

Цей тип ітератора має особливий стан «ітератор кінця потоку», який настає, коли не вдається виконати операцію вводу (яка повертає `fail` після операції з пов'язаним потоком), а також є результатом побудованого за умовчанням об'єкта.

Приклад:

```

// istream_iterator example
#include <iostream>      // std::cin, std::cout
#include <iterator>      // std::istream_iterator

int main()
{
    double value1, value2;
    std::cout << "Please, insert two values: ";

    std::istream_iterator<double> eos;           // end-of-stream iterator
    std::istream_iterator<double> iit(std::cin); // stdin iterator

    if (iit != eos) value1 = *iit;

    ++iit;
    if (iit != eos) value2 = *iit;

    std::cout << value1 << "*" << value2 << "=" << (value1 * value2) << '\n';
    // Please, insert two values: 2 32
    // 2 * 32 = 64

    return 0;
}

```

## ostream\_iterator

```

template <class T, class charT = char, class traits = std::char_traits<charT> >
class ostream_iterator;

```

Ітератори вихідного потоку – це ітератори виводу, які послідовно записують у вихідний потік (наприклад, `cout`). Вони будуються з об'єкта класу `basic_ostream`, з яким вони

стають асоційованими, так що всякий раз, коли виконується операція присвоєння (=) на `ostream_iterator` (розіменованого чи ні), він вставляє новий елемент до потоку.

За бажанням, конструктору можна вказати відокремлювач. Цей відокремлювач записується в потік після вставки кожного елемента.

Приклад:

```
// ostream_iterator example
#include <iostream>      // std::cout
#include <iterator>      // std::ostream_iterator
#include <vector>        // std::vector
#include <algorithm>     // std::copy

int main()
{
    std::vector<int> myvector;
    for (int i = 1; i < 10; ++i) myvector.push_back(i * 10);

    std::ostream_iterator<int> out_it(std::cout, ", ");
    std::copy(myvector.begin(), myvector.end(), out_it);
    // 10, 20, 30, 40, 50, 60, 70, 80, 90,

    return 0;
}
```

### **istreambuf\_iterator**

```
template <class charT, class traits = std::char_traits<charT> >
    class istreambuf_iterator;
```

`istreambuf_iterator` – це вхідні ітератори, які зчитують послідовні елементи з буфера потоку. Вони побудовані з об'єкта `basic_streambuf`, відкритого для зчитування, до якого вони приєднуються.

Цей тип ітератора має особливий стан «ітератор кінця потоку», який встановлюється, коли буде досягнуто кінець потоку, а також є результатом побудованого за замовчуванням об'єкта: це значення можна використовувати як кінець діапазону в будь-якій функції, що приймає діапазони ітераторів, щоб вказати, що діапазон включає всі елементи до кінця вхідного буфера.

Приклад:

```
// istreambuf_iterator example
#include <iostream>      // std::cin, std::cout
#include <iterator>      // std::istreambuf_iterator
#include <string>         // std::string

int main()
{
    std::istreambuf_iterator<char> eos; // end-of-range iterator
    std::istreambuf_iterator<char> iit(std::cin.rdbuf()); // stdin iterator
    std::string mystring;

    std::cout << "Please, enter your name: ";

    while (iit != eos && *iit != '\n') mystring += *iit++;
}
```

```

std::cout << "Your name is " << mystring << ".\n";
// Please, enter your name: HAL 9000
// Your name is HAL 9000.

return 0;
}

```

## ostreambuf\_iterator

```

template <class charT, class traits = std::char_traits<charT> >
class ostreambuf_iterator;

```

ostreambuf\_iterator – це ітератори виводу, які послідовно записують у буфер потоку.

Вони будуються з об'єкта basic\_streambuf, відкритого для запису, з яким вони стають асоційованими.

Приклад:

```

// ostreambuf_iterator example
#include <iostream>      // std::cin, std::cout
#include <iterator>      // std::ostreambuf_iterator
#include <string>        // std::string
#include <algorithm>     // std::copy

int main()
{
    std::string mystring("Some text here...\n");
    std::ostreambuf_iterator<char> out_it(std::cout); // stdout iterator

    std::copy(mystring.begin(), mystring.end(), out_it);
    // Some text here...

    return 0;
}

```

## Алгоритми

Файл заголовку `<algorithm>` визначає набір функцій, спеціально розроблених для використання на діапазонах елементів.

Діапазон – це будь-яка послідовність об'єктів, до якої можна отримати доступ за допомогою ітераторів або вказівників, таких як масив чи екземпляр деяких контейнерів STL.

**Зауваження:** алгоритми працюють за допомогою ітераторів безпосередньо над значеннями, не впливаючи жодним чином на структуру будь-якого можливого контейнера (тобто, ніколи не впливають на розмір або розподіл пам'яті для контейнера).

Всі функції в бібліотеці `<algorithm>` поділяються на наступні категорії:

- Ті, які не модифікують послідовність
- Ті, які модифікують послідовність
- Розбиття
- Сортвання
- Бінарний пошук
- Злиття
- Максимальна купа
- Min / max
- Лексикографічні операції

### Операції, які не модифікують послідовність

<code>accumulate</code>	Обчислити суму елементів діапазону
<code>all_of</code>	Перевірка, чи всі елементи відповідають умові
<code>any_of</code>	Перевірка, чи хоча б один елемент відповідає умові
<code>none_of</code>	Перевірка, чи жоден елемент не відповідає умові
<code>for_each</code>	Застосувати функцію до всіх елементів діапазону
<code>find</code>	Знайти значення в діапазоні
<code>find_if</code>	Знайти елемент в діапазоні, який задовольняє умові
<code>find_if_not</code>	Знайти елемент у діапазоні, який не задовольняє умові
<code>find_end</code>	Знайти останню підпослідовність у діапазоні
<code>find_first_of</code>	Знайти елемент із заданого набору в діапазоні
<code>adjacent_find</code>	Знайти рівні сусідні елементи в діапазоні
<code>count</code>	Обчислити кількість входжень заданого значення в діапазоні
<code>count_if</code>	Повернути кількість елементів у діапазоні, що задовольняє умові
<code>mismatch</code>	Повернути першу позицію, де два діапазони відрізняються
<code>equal</code>	Перевірка, чи збігаються елементи в двох діапазонах
<code>is_permutation</code>	Перевірка, чи діапазон є перестановкою іншого
<code>search</code>	Пошук входження під-діапазону у діапазоні. Шукає діапазон <code>[first1, last1)</code> для першого входження послідовності, визначеної <code>[first2, last2)</code> , і повертає ітератор на його перший елемент, або <code>last1</code> , якщо входжень не знайдено.
<code>search_n</code>	Пошук під-діапазону заданих елементів



## accumulate

Визначено в файлі `<numeric>` (поч. з версії C++20)

```
template< class InputIt, class T >
T accumulate(InputIt first, InputIt last, T init);
```

Обчислює суму елементів у діапазоні `[first, last)`. `init` – початкове значення суми (тобто, обчислюється сума `init` та всіх елементів діапазону).

Приклад:

```
// accumulate example
#include <iostream>
#include <vector>
#include <numeric>

int main()
{
    std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    int sum = std::accumulate(v.begin(), v.end(), 0);

    std::cout << sum << '\n'; // 55
}
```

## all\_of

```
template <class InputIterator, class UnaryPredicate>
bool all_of(InputIterator first, InputIterator last, UnaryPredicate pred);
```

Перевіряє умову для всіх елементів діапазону. Повертає `true`, якщо `pred` повертає `true` для всіх елементів у діапазоні `[first, last)` або якщо діапазон порожній, і `false` в іншому випадку.

Приклад:

```
// all_of example
#include <iostream> // std::cout
#include <algorithm> // std::all_of
#include <array> // std::array

int main()
{
    std::array<int, 8> foo = { 3,5,7,11,13,17,19,23 };

    if (std::all_of(foo.begin(), foo.end(), [](int i) {return i % 2; }))
        std::cout << "All the elements are odd numbers.\n";
    // All the elements are odd numbers.

    return 0;
}
```

## any\_of

```
template <class InputIterator, class UnaryPredicate>
bool any_of(InputIterator first, InputIterator last, UnaryPredicate pred);
```

Перевіряє, чи хоча б якийсь елемент в діапазоні відповідає умові. Повертає `true`, якщо `pred` повертає `true` для будь-якого з елементів у діапазоні `[first, last)`, та `false` в іншому випадку. Якщо `[first, last)` – порожній діапазон, функція повертає `false`.

Приклад:

```
// any_of example
#include <iostream>      // std::cout
#include <algorithm>     // std::any_of
#include <array>         // std::array

int main()
{
    std::array<int, 7> foo = { 0,1,-1,3,-3,5,-5 };

    if (std::any_of(foo.begin(), foo.end(), [](int i) {return i < 0; }))
        std::cout << "There are negative elements in the range.\n";
    // There are negative elements in the range.

    return 0;
}
```

## `none_of`

```
template <class InputIterator, class UnaryPredicate>
bool none_of(InputIterator first, InputIterator last, UnaryPredicate pred);
```

Перевіряє, чи жоден елемент не відповідає умові.

Повертає `true`, якщо `pred` повертає `false` для всіх елементів у діапазоні `[first, last)` або якщо діапазон порожній, і `false` в іншому випадку.

Приклад:

```
// none_of example
#include <iostream>      // std::cout
#include <algorithm>     // std::none_of
#include <array>         // std::array

int main()
{
    std::array<int, 8> foo = { 1,2,4,8,16,32,64,128 };

    if (std::none_of(foo.begin(), foo.end(), [](int i) {return i < 0; }))
        std::cout << "There are no negative elements in the range.\n";
    // There are no negative elements in the range.

    return 0;
}
```

## `for_each`

```
template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function fn);
```

Застосовує функцію `fn` до кожного з елементів у діапазоні `[first, last)`.

Приклад:

```
// for_each example
#include <iostream>      // std::cout
#include <algorithm>     // std::for_each
```

```

#include <vector>           // std::vector

void myfunction(int i)     // function:
{
    std::cout << ' ' << i;
}

struct myclass             // function object type:
{
    void operator() (int i) { std::cout << ' ' << i; }
} myobject;

int main()
{
    std::vector<int> myvector;
    myvector.push_back(10);
    myvector.push_back(20);
    myvector.push_back(30);

    std::cout << "myvector contains:";
    for_each(myvector.begin(), myvector.end(), myfunction);
    std::cout << '\n';

    // or:
    std::cout << "myvector contains:";
    for_each(myvector.begin(), myvector.end(), myobject);
    std::cout << '\n';
    // myvector contains: 10 20 30
    // myvector contains: 10 20 30

    return 0;
}

```

## find

```

template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& val);

```

Повертає ітератор до першого елемента в діапазоні [first,last), який дорівнює значенню val. Якщо такого елемента не знайдено, функція повертає last.

Функція використовує `operator==` для порівняння окремих елементів із val.

Приклад:

```

// find example
#include <iostream>         // std::cout
#include <algorithm>        // std::find
#include <vector>           // std::vector

int main()
{
    // using std::find with array and pointer:
    int myints[] = { 10, 20, 30, 40 };
    int* p;

    p = std::find(myints, myints + 4, 30);
    if (p != myints + 4)
        std::cout << "Element found in myints: " << *p << '\n';
    else
        std::cout << "Element not found in myints\n";
}

```

```

// using std::find with vector and iterator:
std::vector<int> myvector(myints, myints + 4);
std::vector<int>::iterator it;

it = find(myvector.begin(), myvector.end(), 30);
if (it != myvector.end())
    std::cout << "Element found in myvector: " << *it << '\n';
else
    std::cout << "Element not found in myvector\n";
// Element found in myints: 30
// Element found in myvector : 30

return 0;
}

```

## find\_if

```

template <class InputIterator, class UnaryPredicate>
InputIterator find_if(InputIterator first, InputIterator last,
                    UnaryPredicate pred);

```

Повертає ітератор до першого елемента в діапазоні [first, last), для якого pred повертає true. Якщо такого елемента не знайдено, функція повертає last.

Приклад:

```

// find_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::find_if
#include <vector>        // std::vector

bool IsOdd(int i)
{
    return ((i % 2) == 1);
}

int main()
{
    std::vector<int> myvector;

    myvector.push_back(10);
    myvector.push_back(25);
    myvector.push_back(40);
    myvector.push_back(55);

    std::vector<int>::iterator it
        = std::find_if(myvector.begin(), myvector.end(), IsOdd);
    std::cout << "The first odd value is " << *it << '\n';
    // The first odd value is 25

    return 0;
}

```

## find\_if\_not

```

template <class InputIterator, class UnaryPredicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                        UnaryPredicate pred);

```

Повертає ітератор до першого елемента в діапазоні [first, last), для якого pred повертає false. Якщо такого елемента не знайдено, функція повертає last.

Приклад:

```
// find_if_not example
#include <iostream>      // std::cout
#include <algorithm>      // std::find_if_not
#include <array>          // std::array

int main()
{
    std::array<int, 5> foo = { 1,2,3,4,5 };

    std::array<int, 5>::iterator it =
        std::find_if_not(foo.begin(), foo.end(), [](int i) {return i % 2; });
    std::cout << "The first even value is " << *it << '\n';
    // The first even value is 2

    return 0;
}
```

## find\_end

```
// equality (1)
template <class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2);

// predicate (2)
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2,
                              BinaryPredicate pred);
```

Шукає останню підпоследовність в діапазоні.

Шукає діапазон [first1, last1) для останнього входження послідовності, визначеного [first2, last2), і повертає ітератор до його першого елемента, або last1, якщо входження не знайдено.

Елементи в обох діапазонах порівнюються послідовно за допомогою `operator==` (або `pred`, у версії (2)): підпоследовність діапазону [first1, last1) вважається збігом лише тоді, коли результат буде `true` для всіх елементів [first2, last2).

Ця функція повертає останнє з таких входжень. Алгоритм, який замість цього повертає перше входження – `search`.

Приклад:

```
// find_end example
#include <iostream>      // std::cout
#include <algorithm>      // std::find_end
#include <vector>         // std::vector

bool myfunction(int i, int j)
{
    return (i == j);
}

int main()
{
    int myints[] = { 1,2,3,4,5,1,2,3,4,5 };
}
```

```

std::vector<int> haystack(myints, myints + 10);

int needle1[] = { 1,2,3 };

// using default comparison:
std::vector<int>::iterator it;
it = std::find_end(haystack.begin(), haystack.end(), needle1, needle1 + 3);

if (it != haystack.end())
    std::cout << "needle1 last found at position "
               << (it - haystack.begin()) << '\n';

int needle2[] = { 4,5,1 };

// using predicate comparison:
it = std::find_end(haystack.begin(), haystack.end(),
                  needle2, needle2 + 3, myfunction);

if (it != haystack.end())
    std::cout << "needle2 last found at position "
               << (it - haystack.begin()) << '\n';
// needle1 last found at position 5
// needle2 last found at position 3

return 0;
}

```

## find\_first\_of

```

// equality (1)
template <class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
                                   ForwardIterator2 first2, ForwardIterator2 last2);

// predicate (2)
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,
                                   ForwardIterator2 first2, ForwardIterator2 last2,
                                   BinaryPredicate pred);

```

Шукає елемент із набору в діапазоні.

Повертає ітератор до першого елемента в діапазоні [first1, last1), який відповідає будь-якому з елементів у [first2, last2). Якщо такого елемента не знайдено, функція повертає last1. Елементи у [first1, last1) послідовно порівнюються з кожним із значень у [first2, last2) за допомогою `operator==` (або `pred`, у версії (2)), до тих пір, поки не буде виявлено збігу.

Приклад:

```

// find_first_of example
#include <iostream>      // std::cout
#include <algorithm>     // std::find_first_of
#include <vector>        // std::vector
#include <cctype>        // std::tolower

bool comp_case_insensitive(char c1, char c2)
{
    return (std::tolower(c1) == std::tolower(c2));
}

```

```

int main()
{
    int mychars[] = { 'a','b','c','A','B','C' };
    std::vector<char> haystack(mychars, mychars + 6);
    std::vector<char>::iterator it;

    int needle[] = { 'A','B','C' };

    // using default comparison:
    it = find_first_of(haystack.begin(), haystack.end(), needle, needle + 3);

    if (it != haystack.end())
        std::cout << "The first match is: " << *it << '\n';

    // using predicate comparison:
    it = find_first_of(haystack.begin(), haystack.end(),
                      needle, needle + 3, comp_case_insensitive);

    if (it != haystack.end())
        std::cout << "The first match is: " << *it << '\n';
    // The first match is: A
    // The first match is: a

    return 0;
}

```

## adjacent\_find

```

// equality (1)
template <class ForwardIterator>
    ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

// predicate (2)
template <class ForwardIterator, class BinaryPredicate>
    ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);

```

Знаходить однакові сусідні елементи в діапазоні.

Шукає в діапазоні [first, last) перше входження двох послідовних елементів, що збігаються.

Повертає ітератор до першого з цих двох елементів, або last, якщо такої пари не знайдено.

Два елементи збігаються, якщо вони однакові при порівнянні за допомогою `operator==` (або використовуючи `pred`, у версії (2)).

Приклад:

```

// adjacent_find example
#include <iostream>      // std::cout
#include <algorithm>     // std::adjacent_find
#include <vector>        // std::vector

bool myfunction(int i, int j)
{
    return (i == j);
}

```

```

int main()
{
    int myints[] = { 5,20,5,30,30,20,10,10,20 };
    std::vector<int> myvector(myints, myints + 8);
    std::vector<int>::iterator it;

    // using default comparison:
    it = std::adjacent_find(myvector.begin(), myvector.end());

    if (it != myvector.end())
        std::cout << "the first pair of repeated elements are: " << *it << '\n';

    //using predicate comparison:
    it = std::adjacent_find(++it, myvector.end(), myfunction);

    if (it != myvector.end())
        std::cout << "the second pair of repeated elements are: " << *it << '\n';

    // the first pair of repeated elements are: 30
    // the second pair of repeated elements are: 10

    return 0;
}

```

## count

```

template <class InputIterator, class T>
typename std::iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& val);

```

Кількість входжень значення в діапазоні.

Повертає кількість елементів у діапазоні [first, last), які дорівнюють значенню val.

Функція використовує `operator==` для порівняння окремих елементів із val.

Приклад:

```

// count algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::count
#include <vector>        // std::vector

int main()
{
    // counting elements in array:
    int myints[] = { 10,20,30,30,20,10,10,20 }; // 8 elements
    int mycount = std::count(myints, myints + 8, 10);
    std::cout << "10 appears " << mycount << " times.\n";

    // counting elements in container:
    std::vector<int> myvector(myints, myints + 8);
    mycount = std::count(myvector.begin(), myvector.end(), 20);
    std::cout << "20 appears " << mycount << " times.\n";
    // 10 appears 3 times.
    // 20 appears 3 times.

    return 0;
}

```



## count\_if

```
template <class InputIterator, class UnaryPredicate>
    typename std::iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, UnaryPredicate pred);
```

Повертає кількість елементів у діапазоні, які відповідають умові.

Повертає кількість елементів у діапазоні [first, last), для яких pred має значення true.

Приклад:

```
// count_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::count_if
#include <vector>        // std::vector

bool IsOdd(int i) { return ((i % 2) == 1); }

int main()
{
    std::vector<int> myvector;
    for (int i = 1; i < 10; i++) myvector.push_back(i);
    // myvector: 1 2 3 4 5 6 7 8 9

    int mycount = count_if(myvector.begin(), myvector.end(), IsOdd);
    std::cout << "myvector contains " << mycount << " odd values.\n";
    // myvector contains 5 odd values.

    return 0;
}
```

## mismatch

```
// equality (1)
template <class InputIterator1, class InputIterator2>
    std::pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

// predicate (2)
template <class InputIterator1, class InputIterator2, class BinaryPredicate>
    std::pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
```

Повертає першу позицію, у якій два діапазони відрізняються

Порівнює елементи в діапазоні [first1, last1) з відповідними елементами в діапазоні, що починається з first2, і повертає позицію першого елемента обох послідовностей, в якій вони не збігаються.

Елементи порівнюються за допомогою operator== (або pred, у версії (2)).

Функція повертає об'єкт pair для пари ітераторів до перших елементів кожного діапазону, у яких вони не збігаються.

Приклад:

```

// mismatch algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::mismatch
#include <vector>        // std::vector
#include <utility>       // std::pair

bool mypredicate(int i, int j)
{
    return (i == j);
}

int main()
{
    std::vector<int> myvector;
    for (int i = 1; i < 6; i++) myvector.push_back(i * 10);
    // myvector: 10 20 30 40 50

    int myints[] = { 10,20,80,320,1024 };
    // myints: 10 20 80 320 1024

    std::pair<std::vector<int>::iterator, int*> mypair;

    // using default comparison:
    mypair = std::mismatch(myvector.begin(), myvector.end(), myints);
    std::cout << "First mismatching elements: " << *mypair.first;
    std::cout << " and " << *mypair.second << '\n';

    ++mypair.first; ++mypair.second;

    // using predicate comparison:
    mypair = std::mismatch(mypair.first, myvector.end(), mypair.second, mypredicate);
    std::cout << "Second mismatching elements: " << *mypair.first;
    std::cout << " and " << *mypair.second << '\n';
    // First mismatching elements: 30 and 80
    // Second mismatching elements : 40 and 320

    return 0;
}

```

## equal

```

// equality (1)
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

// predicate (2)
template <class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate pred);

```

Перевіряє, чи елементи в двох діапазонах збігаються.

Порівнює елементи в діапазоні [first1,last1) з елементами в діапазоні, що починається з first2, і повертає **true**, якщо всі відповідні елементи в обох діапазонах збігаються.

Елементи порівнюються за допомогою **operator==** (або **pred**, у версії (2)).

Приклад:

```

// equal algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::equal
#include <vector>        // std::vector

bool mypredicate(int i, int j)
{
    return (i == j);
}

int main()
{
    int myints[] = { 20,40,60,80,100 };           // myints: 20 40 60 80 100
    std::vector<int> myvector(myints, myints + 5); // myvector: 20 40 60 80 100

    // using default comparison:
    if (std::equal(myvector.begin(), myvector.end(), myints))
        std::cout << "The contents of both sequences are equal.\n";
    else
        std::cout << "The contents of both sequences differ.\n";

    myvector[3] = 81;                             // myvector: 20 40 60 81 100

    // using predicate comparison:
    if (std::equal(myvector.begin(), myvector.end(), myints, mypredicate))
        std::cout << "The contents of both sequences are equal.\n";
    else
        std::cout << "The contents of both sequences differ.\n";
    // The contents of both sequences are equal.
    // The contents of both sequence differ.

    return 0;
}

```

## is\_permutation

```

// equality (1)
template <class ForwardIterator1, class ForwardIterator2>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2);

// predicate (2)
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, BinaryPredicate pred);

```

Перевіряє, чи діапазон є перестановкою іншого.

Порівнює елементи в діапазоні [first1,last1) з елементами в діапазоні, що починається з first2, і повертає **true**, якщо всі елементи в обох діапазонах збігаються, навіть якщо вони розташовані в іншому порядку.

Елементи порівнюються за допомогою **operator==** (або pred, у версії (2)).

Приклад:

```

// is_permutation example
#include <iostream>      // std::cout
#include <algorithm>     // std::is_permutation
#include <array>         // std::array

```

```
int main()
{
    std::array<int, 5> foo = { 1,2,3,4,5 };
    std::array<int, 5> bar = { 3,1,4,5,2 };

    if (std::is_permutation(foo.begin(), foo.end(), bar.begin()))
        std::cout << "foo and bar contain the same elements.\n";
    // foo and bar contain the same elements.

    return 0;
}
```

## search

```
// equality (1)
template <class ForwardIterator1, class ForwardIterator2>
    ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2);

// predicate (2)
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
    ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           BinaryPredicate pred);
```

Шукає входження під-діапазону у діапазоні.

Шукає в діапазоні [first1, last1) перше входження послідовності, визначеної [first2, last2), і повертає ітератор до його першого елемента, або last1, якщо входжень не знайдено. Елементи в обох діапазонах порівнюються послідовно за допомогою `operator==` (або `pred`, у версії (2)): підпослідовність діапазону [first1, last1) вважається збігом лише тоді, коли результат буде `true` для всіх елементів [first2, last2).

Ця функція повертає перше з таких входжень. Алгоритм, який замість цього повертає останнє входження – `find_end`.

Приклад:

```
// search algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::search
#include <vector>        // std::vector

bool mypredicate(int i, int j)
{
    return (i == j);
}

int main()
{
    std::vector<int> haystack;

    // set some values:      haystack: 10 20 30 40 50 60 70 80 90
    for (int i = 1; i < 10; i++) haystack.push_back(i * 10);

    // using default comparison:
    int needle1[] = { 40,50,60,70 };

    std::vector<int>::iterator it;
```

```

it = std::search(haystack.begin(), haystack.end(), needle1, needle1 + 4);

if (it != haystack.end())
    std::cout << "needle1 found at position " << (it - haystack.begin()) << '\n';
else
    std::cout << "needle1 not found\n";

// using predicate comparison:
int needle2[] = { 20,30,50 };
it = std::search(haystack.begin(), haystack.end(),
                needle2, needle2 + 3, mypredicate);

if (it != haystack.end())
    std::cout << "needle2 found at position " << (it - haystack.begin()) << '\n';
else
    std::cout << "needle2 not found\n";

// needle1 found at position 3
// needle2 not found

return 0;
}

```

## search\_n

```

// equality (1)
template <class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                        Size count, const T& val);

// predicate (2)
template <class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                        Size count, const T& val, BinaryPredicate pred);

```

Шукає під-діапазон заданих елементів.

Шукає в діапазоні [first, last) послідовність з count елементів, кожен з яких дорівнює значенню val (або для якого pred повертає true).

Функція повертає ітератор до першого з таких елементів або last, якщо такої послідовності не знайдено.

Приклад:

```

// search_n example
#include <iostream> // std::cout
#include <algorithm> // std::search_n
#include <vector> // std::vector

bool mypredicate(int i, int j)
{
    return (i == j);
}

int main()
{
    int myints[] = { 10,20,30,30,20,10,10,20 };
    std::vector<int> myvector(myints, myints + 8);

    std::vector<int>::iterator it;
}

```

```

// using default comparison:
it = std::search_n(myvector.begin(), myvector.end(), 2, 30);

if (it != myvector.end())
    std::cout << "two 30s found at position " << (it - myvector.begin()) << '\n';
else
    std::cout << "match not found\n";

// using predicate comparison:
it = std::search_n(myvector.begin(), myvector.end(), 2, 10, mypredicate);

if (it != myvector.end())
    std::cout << "two 10s found at position "
                << int(it - myvector.begin()) << '\n';
else
    std::cout << "match not found\n";
// Two 30s found at position 2
// Two 10s found at position 5

return 0;
}

```

## Операції, які модифікують послідовність

copy	Копіювати діапазон елементів
copy_n	Копіювати перші n елементів
copy_if	Копіювати певні елементи діапазону
copy_backward	Копіювати діапазон елементів назад
move	Переміщення діапазону елементів
move_backward	Перемістити діапазон елементів назад
swap	Обмін значеннями двох об'єктів
swap_ranges	Обмін значеннями двох діапазонів
iter_swap	Обмін значеннями об'єктів, на які вказують два ітератори
transform	Діапазон перетворення
replace	Замінити значення в діапазоні
replace_if	Замінити значення в діапазоні, які задовольняють умові
replace_copy	Копіювати замінене значення діапазону
replace_copy_if	Копіювати замінені значення діапазону, які задовольняють умові
fill	Заповнити діапазон значенням
fill_n	Заповнити послідовність значенням
generate	Створення значень для діапазону за допомогою функції
generate_n	Створення значень для послідовності за допомогою функції
remove	Видалити значення з діапазону
remove_if	Видалити елементи з діапазону, які задовольняють умові
remove_copy	Копіювати діапазон за винятком вказаного значення
remove_copy_if	Копіювати діапазон за винятком значень, які задовольняють умові
unique	Видалити послідовні дублікати в діапазоні
unique_copy	Копіювати діапазон, видаляючи дублікати
reverse	Змінити порядок розташування елементів на зворотній
reverse_copy	Копіювати елементи із заміною порядку розташування на зворотній
rotate	Циклічний зсув вліво елементів в діапазоні
rotate_copy	Копіювати елементи з циклічним зсувом вліво
random_shuffle	Випадкове переставлення елементів у діапазоні
shuffle	Випадкове переставлення елементів у діапазоні за допомогою генератора

### copy

```
template <class InputIterator, class OutputIterator>
    OutputIterator copy(InputIterator first, InputIterator last,
                       OutputIterator result);
```

Копіює діапазон елементів.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result.

Функція повертає ітератор до кінця діапазону призначення (який вказує на елемент, що слідує за останнім скопійованим елементом).

Діапазони не повинні перекриватися: тобто, result не може вказувати на елемент у діапазоні [first, last). Для таких випадків див. copy\_backward.

Приклад:

```

// copy algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::copy
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10,20,30,40,50,60,70 };
    std::vector<int> myvector(7);

    std::copy(myints, myints + 7, myvector.begin());

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 10 20 30 40 50 60 70

    return 0;
}

```

## copy\_n

```

template <class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);

```

Копіює перші *n* елементів з діапазону, що починається з *first*, в діапазон, що починається з *result*. Функція повертає ітератор до кінця діапазону призначення (який вказує на елемент після останнього скопійованого елементу).

Якщо *n* від'ємне, функція нічого не робить.

Якщо діапазони перекриваються, деякі елементи в діапазоні, вказаному *result*, можуть мати невизначені, але допустимі значення.

Приклад:

```

// copy_n algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::copy
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10,20,30,40,50,60,70 };
    std::vector<int> myvector;

    myvector.resize(7);    // allocate space for 7 elements

    std::copy_n(myints, 7, myvector.begin());

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 10 20 30 40 50 60 70

    return 0;
}

```



## copy\_if

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
                      OutputIterator result, UnaryPredicate pred);
```

Копіює певні елементи діапазону. Копіює елементи в діапазоні [first, last), для яких pred повертає true, в діапазон, що починається з result.

Приклад:

```
// copy_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::copy_if, std::distance
#include <vector>        // std::vector

int main()
{
    std::vector<int> foo = { 25,15,5,-5,-15 };
    std::vector<int> bar(foo.size());

    // copy only positive numbers:
    auto it = std::copy_if(foo.begin(), foo.end(), bar.begin(),
                          [](int i) {return i > 0; }); // lambda

    bar.resize(std::distance(bar.begin(), it)); // shrink container to new size

    std::cout << "bar contains:";
    for (int& x : bar) std::cout << ' ' << x;
    std::cout << '\n'; // bar contains: 25 15 5

    return 0;
}
```

## copy\_backward

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                    BidirectionalIterator1 last,
                                    BidirectionalIterator2 result);
```

Копіює діапазон елементів «назад» – в позицію, яка визначається кінцем діапазону призначення.

Копіює елементи в діапазоні [first, last), починаючи з кінця, в діапазон, що закінчується на result.

Функція повертає ітератор до першого елемента в діапазоні призначення.

Результуючий діапазон буде містити елементи в точно такому ж порядку, як [first, last). Щоб змінити порядок, див. reverse\_copy.

Функція починає з копіювання \*(last-1) у \*(result-1), а потім йде назад по елементах, що передують цим, до досягнення first (і включно з ним).

Діапазони не можуть перекриватися: тобто, result (який є елементом після-останнього у діапазоні призначення) не може вказувати на елемент у діапазоні (first, last]. Для таких випадків див. copy.

Приклад:

```
// copy_backward example
#include <iostream>      // std::cout
#include <algorithm>     // std::copy_backward
#include <vector>        // std::vector

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i <= 5; i++)
        myvector.push_back(i * 10);          // myvector: 10 20 30 40 50

    myvector.resize(myvector.size() + 3);    // allocate space for 3 more elements

    std::copy_backward(myvector.begin(), myvector.begin() + 5, myvector.end());

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 10 20 30 10 20 30 40 50

    return 0;
}
```

## move

```
template <class InputIterator, class OutputIterator>
OutputIterator move(InputIterator first, InputIterator last,
                   OutputIterator result);
```

Переміщення діапазону елементів.

Переміщує елементи в діапазоні [first, last) в діапазон, що починається з result.

Значення елементів у [first, last) передається елементам, на які вказує result. Після виклику елементи в діапазоні [first, last) залишаються у невизначеному, але дійсному стані.

Діапазони не повинні перекриватися таким чином, щоб result вказував на елемент у діапазоні [first, last).

Для таких випадків див. move\_backward.

Приклад:

```
// move algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::move (ranges)
#include <utility>       // std::move (objects)
#include <vector>        // std::vector
#include <string>        // std::string

int main()
{
    std::vector<std::string> foo = { "air", "water", "fire", "earth" };
    std::vector<std::string> bar(4);
```

```

// moving ranges:
std::cout << "Moving ranges...\n";
std::move(foo.begin(), foo.begin() + 4, bar.begin());

std::cout << "foo contains " << foo.size() << " elements:";
std::cout << " (each in an unspecified but valid state)";
std::cout << '\n';

std::cout << "bar contains " << bar.size() << " elements:";
for (std::string& x : bar) std::cout << " [" << x << "]\n";

// moving container:
std::cout << "Moving container...\n";
foo = std::move(bar);

std::cout << "foo contains " << foo.size() << " elements:";
for (std::string& x : foo) std::cout << " [" << x << "]\n";

std::cout << "bar is in an unspecified but valid state";
std::cout << '\n';
// Moving ranges...
// foo contains 4 elements: (each in an unspecified but valid state)
// bar contains 4 elements : [air] [water] [fire] [earth]
// Moving container...
// foo contains 4 elements : [air] [water] [fire] [earth]
// bar is in an unspecified but valid state

return 0;
}

```

## move\_backward

```

template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 move_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result);

```

Перемістити діапазон елементів «назад» — в позицію, яка визначається кінцем діапазону призначення.

Переміщує елементи в діапазоні `[first, last)`, починаючи з кінця, в діапазон, що закінчується на `result`.

Функція повертає ітератор до першого елемента в діапазоні призначення.

Результуючий діапазон буде містити елементи в точно такому ж порядку, як `[first, last)`. Щоб змінити порядок, див. `reverse`.

Функція починає з переміщення `*(last-1)` у `*(result-1)`, а потім йде назад по елементах, що передують цим, до досягнення `first` (і включно з ним).

Діапазони не можуть перекриватися: тобто, `result` (який є елементом після-останнього у діапазоні призначення) не може вказувати на елемент у діапазоні `(first, last]`. Для таких випадків див. `move`.

Приклад:

```
// move_backward example
#include <iostream>      // std::cout
#include <algorithm>     // std::move_backward
#include <string>        // std::string

int main()
{
    std::string elems[10] = { "air", "water", "fire", "earth" };

    // insert new element at the beginning:
    std::move_backward(elems, elems + 4, elems + 5);
    elems[0] = "ether";

    std::cout << "elems contains:";
    for (int i = 0; i < 10; ++i)
        std::cout << " [" << elems[i] << " ]";
    std::cout << '\n';
    // elems contains: [ether] [air] [water] [fire] [earth] [] [] [] [] []

    return 0;
}
```

## swap

```
template <class T>
void swap(T& a, T& b);
```

Обмін значеннями двох об'єктів. Обмінює значення a і b.

Приклад:

```
// swap algorithm example (C++98)
#include <iostream>      // std::cout
#include <algorithm>     // std::swap
#include <vector>        // std::vector

int main()
{
    int x = 10, y = 20;                // x:10 y:20
    std::swap(x, y);                  // x:20 y:10

    std::vector<int> foo(4, x), bar(6, y); // foo:4x20 bar:6x10
    std::swap(foo, bar);              // foo:6x10 bar:4x20

    std::cout << "foo contains:";
    for (std::vector<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // foo contains: 10 10 10 10 10 10

    return 0;
}
```

## swap\_ranges

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

Обмін значень двох діапазонів.

Обмінює значення кожного з елементів у діапазоні [first1, last1) зі значеннями відповідних елементів у діапазоні, що починається з first2.

Функція викликає swap для обміну елементів.

Приклад:

```
// swap_ranges example
#include <iostream>      // std::cout
#include <algorithm>     // std::swap_ranges
#include <vector>        // std::vector

int main()
{
    std::vector<int> foo(5, 10);          // foo: 10 10 10 10 10
    std::vector<int> bar(5, 33);         // bar: 33 33 33 33 33

    std::swap_ranges(foo.begin() + 1, foo.end() - 1, bar.begin());

    // print out results of swap:
    std::cout << "foo contains:";
    for (std::vector<int>::iterator it = foo.begin(); it != foo.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    std::cout << "bar contains:";
    for (std::vector<int>::iterator it = bar.begin(); it != bar.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // foo contains: 10 33 33 33 10
    // bar contains: 10 10 10 33 33

    return 0;
}
```

## iter\_swap

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Обмін значень об'єктів, на які вказують два ітератори.

Міняє місцями елементи, на які вказують a та b.

Функція викликає swap для обміну елементів.

Приклад:

```
// iter_swap example
#include <iostream>      // std::cout
#include <algorithm>     // std::iter_swap
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10,20,30,40,50 };          // myints:    10  20  30  40  50
    std::vector<int> myvector(4, 99);          // myvector: 99  99  99  99

    std::iter_swap(myints, myvector.begin());  // myints:   [99] 20  30  40  50
                                              // myvector: [10] 99  99  99

    std::iter_swap(myints + 3, myvector.begin() + 2); // myints:    99  20  30 [99] 50
                                                         // myvector: 10  99 [40] 99
}
```

```

std::cout << "myvector contains: ";
for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// myvector contains: 10 99 40 99

return 0;
}

```

## transform

```

// unary operation (1)
template <class InputIterator, class OutputIterator, class UnaryOperation>
    OutputIterator transform(InputIterator first1, InputIterator last1,
                            OutputIterator result, UnaryOperation op);

// binary operation (2)
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation>
    OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, OutputIterator result,
                            BinaryOperation binary_op);

```

Перетворення діапазону.

Послідовно застосовує операцію до елементів одного (1) або двох (2) діапазонів і зберігає результат у діапазоні, який починається з result.

(1) унарна операція.

Застосовує op до кожного з елементів у діапазоні [first1, last1) і зберігає значення, яке повертається кожною операцією, в діапазоні, який починається в result.

(2) бінарна операція.

Викликає binary\_op, використовуючи кожен з елементів у діапазоні [first1, last1) як перший аргумент, а відповідний елемент у діапазоні, який починається з first2, як другий аргумент. Значення, що повертається кожним викликом, зберігається в діапазоні, який починається з result.

Приклад:

```

// transform algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::transform
#include <vector>         // std::vector
#include <functional>    // std::plus

int op_increase(int i) { return ++i; }

int main()
{
    std::vector<int> foo;
    std::vector<int> bar;

    // set some values:
    for (int i = 1; i < 6; i++)
        foo.push_back(i * 10);
}
// foo: 10 20 30 40 50

```

```

bar.resize(foo.size()); // allocate space

std::transform(foo.begin(), foo.end(), bar.begin(), op_increase);
// bar: 11 21 31 41 51

// std::plus adds together its two arguments:
std::transform(foo.begin(), foo.end(), bar.begin(), foo.begin(), std::plus<int>());
// foo: 21 41 61 81 101

std::cout << "foo contains:";
for (std::vector<int>::iterator it = foo.begin(); it != foo.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// foo contains: 21 41 61 81 101

return 0;
}

```

## replace

```

template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);

```

Замінює значення в діапазоні.

Присвоює `new_value` для всіх елементів у діапазоні `[first, last)`, які дорівнюють `old_value`. Функція використовує `operator==` для порівняння окремих елементів із `old_value`.

Приклад:

```

// replace algorithm example
#include <iostream> // std::cout
#include <algorithm> // std::replace
#include <vector> // std::vector

int main()
{
    int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
    std::vector<int> myvector(myints, myints + 8); // 10 20 30 30 20 10 10 20

    std::replace(myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10 10 99

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 10 99 30 30 99 10 10 99

    return 0;
}

```

## replace\_if

```

template <class ForwardIterator, class UnaryPredicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                UnaryPredicate pred, const T& new_value);

```

Замінює значення в діапазоні.

Присвоює new\_value всім елементам у діапазоні [first, last), для яких pred повертає true.

Приклад:

```
// replace_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::replace_if
#include <vector>        // std::vector

bool IsOdd(int i) { return ((i % 2) == 1); }

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; i++) myvector.push_back(i);           // 1 2 3 4 5 6 7 8 9

    std::replace_if(myvector.begin(), myvector.end(), IsOdd, 0); // 0 2 0 4 0 6 0 8 0

    std::cout << "myvector contains: ";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 0 2 0 4 0 6 0 8 0

    return 0;
}
```

## replace\_copy

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                           OutputIterator result,
                           const T& old_value, const T& new_value);
```

Копіює діапазон, замінюючи значення.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result, замінюючи кожне входження old\_value на new\_value. Функція використовує operator== для порівняння окремих елементів із old\_value.

Діапазони не можуть перекриватися, тобто result не може вказувати на елемент у діапазоні [first, last).

Приклад:

```
// replace_copy example
#include <iostream>      // std::cout
#include <algorithm>     // std::replace_copy
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };

    std::vector<int> myvector(8);
    std::replace_copy(myints, myints + 8, myvector.begin(), 20, 99);
}
```



```

std::cout << "myvector contains: ";
for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// myvector contains: 10 99 30 30 99 10 10 99

return 0;
}

```

## replace\_copy\_if

```

template <class InputIterator, class OutputIterator, class UnaryPredicate, class T>
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, UnaryPredicate pred,
                              const T& new_value);

```

Копіює діапазон, замінюючи значення.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result, замінюючи ті, для яких pred повертає true, на new\_value.

Приклад:

```

// replace_copy_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::replace_copy_if
#include <vector>        // std::vector

bool IsOdd(int i) { return ((i % 2) == 1); }

int main()
{
    std::vector<int> foo, bar;

    // set some values:
    for (int i = 1; i < 10; i++) foo.push_back(i);      // 1 2 3 4 5 6 7 8 9

    bar.resize(foo.size()); // allocate space
    std::replace_copy_if(foo.begin(), foo.end(), bar.begin(), IsOdd, 0);
    // 0 2 0 4 0 6 0 8 0

    std::cout << "bar contains: ";
    for (std::vector<int>::iterator it = bar.begin(); it != bar.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // second contains: 0 2 0 4 0 6 0 8 0

    return 0;
}

```

## fill

```

template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& val);

```

Заповнює діапазон вказаним значенням.

Присвоює значення val усім елементам у діапазоні [first, last).

Приклад:

```

// fill algorithm example
#include <iostream>      // std::cout

```

```

#include <algorithm>    // std::fill
#include <vector>       // std::vector

int main()
{
    std::vector<int> myvector(8);           // myvector: 0 0 0 0 0 0 0 0

    std::fill(myvector.begin(), myvector.begin() + 4, 5); // myvector: 5 5 5 5 0 0 0 0
    std::fill(myvector.begin() + 3, myvector.end() - 2, 8); // myvector: 5 5 5 8 8 8 0 0

    std::cout << "myvector contains: ";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 5 5 5 8 8 8 0 0

    return 0;
}

```

## fill\_n

```

template <class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& val);

```

Заповнює послідовність вказаним значенням.

Присвоює значення val першим n елементам послідовності, що починається з first.

Приклад:

```

// fill_n example
#include <iostream>    // std::cout
#include <algorithm>    // std::fill_n
#include <vector>       // std::vector

int main()
{
    std::vector<int> myvector(8, 10);           // myvector: 10 10 10 10 10 10 10 10

    std::fill_n(myvector.begin(), 4, 20);       // myvector: 20 20 20 20 10 10 10 10
    std::fill_n(myvector.begin() + 3, 3, 33);   // myvector: 20 20 20 33 33 33 10 10

    std::cout << "myvector contains: ";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 20 20 20 33 33 33 10 10

    return 0;
}

```

## generate

```

template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);

```

Формує значення для діапазону за допомогою функції.

Присвоює значення, що повертається послідовними викликами gen елементам у діапазоні [first, last).

Приклад:

```

// generate algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::generate
#include <vector>        // std::vector
#include <ctime>         // std::time
#include <cstdlib>       // std::rand, std::srand

// function generator:
int RandomNumber() { return (std::rand() % 100); }

// class generator:
struct c_unique
{
    int current;
    c_unique() { current = 0; }
    int operator()() { return ++current; }
} UniqueNumber;

int main()
{
    std::srand(unsigned(std::time(0)));

    std::vector<int> myvector(8);

    std::generate(myvector.begin(), myvector.end(), RandomNumber);

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    std::generate(myvector.begin(), myvector.end(), UniqueNumber);

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 57 87 76 66 85 54 17 15
    // myvector contains : 1 2 3 4 5 6 7 8

    return 0;
}

```

## generate\_n

```

template <class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);

```

Формує значення для послідовності за допомогою функції. Присвоює значення, які повертають послідовні виклики gen, першим n елементам послідовності, що починається з first.

Приклад:

```

// generate_n example
#include <iostream>      // std::cout
#include <algorithm>     // std::generate_n

int current = 0;
int UniqueNumber() { return ++current; }

```

```

int main()
{
    int myarray[9];

    std::generate_n(myarray, 9, UniqueNumber);

    std::cout << "myarray contains:";
    for (int i = 0; i < 9; ++i)
        std::cout << ' ' << myarray[i];
    std::cout << '\n';
    // myarray contains: 1 2 3 4 5 6 7 8 9

    return 0;
}

```

## remove

```

template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& val);

```

Видаляє значення з діапазону. Перетворює діапазон [first, last) у діапазон, в якому видалені всі елементи, що дорівнюють значенню val, і повертає ітератор на новий кінець цього діапазону. Функція не може змінювати властивості об'єкта, що містить діапазон елементів (тобто, вона не може змінювати розмір масиву або контейнера): видалення здійснюється заміною всіх елементів, які дорівнюють значенню val, наступним елементом, який не дорівнює val. При цьому в якості сигналу про новий розмір скороченого діапазону, функція повертає ітератор до елемента, який слід вважати новим елементом після-кінця.

Відносний порядок елементів, що не видаляються, зберігається, тоді як елементи між результатом ітератором та last залишаються у дійсному, але невизначеному стані.

Функція використовує `operator==` для порівняння окремих елементів із val.

Приклад:

```

// remove algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::remove

int main()
{
    int myints[] = { 10,20,30,30,20,10,10,20 };           // 10 20 30 30 20 10 10 20

    // bounds of range:
    int* pbegin = myints;                                // ^
    int* pend = myints + sizeof(myints) / sizeof(int);   // ^

    pend = std::remove(pbegin, pend, 20);                 // 10 30 30 10 10 ? ? ?
                                                         // ^           ^

    std::cout << "range contains:";
    for (int* p = pbegin; p != pend; ++p)
        std::cout << ' ' << *p;
    std::cout << '\n';
    // range contains: 10 30 30 10 10

    return 0;
}

```

## remove\_if

```
template <class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                          UnaryPredicate pred);
```

Видаляє елементи з діапазону.

Перетворює діапазон [first, last) в діапазон, в якому всі елементи, для яких pred повертає true – видалені, і повертає ітератор до нового кінця цього діапазону.

Функція не може змінювати властивості об'єкта, що містить діапазон елементів (тобто, вона не може змінювати розмір масиву або контейнера). Видалення здійснюється заміною всіх елементів, для яких pred повертає true, наступним елементом, для якого pred не true. В якості сигналу про новий розмір скороченого діапазону, функція повертає ітератор до елемента, який слід вважати новим елементом після-кінця.

Відносний порядок елементів, що не видаляються, зберігається, тоді як елементи між результатом ітератором та last залишаються у дійсному, але невизначеному стані.

Приклад:

```
// remove_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::remove_if

bool IsOdd(int i) { return ((i % 2) == 1); }

int main()
{
    int myints[] = { 1,2,3,4,5,6,7,8,9 };           // 1 2 3 4 5 6 7 8 9

    // bounds of range:
    int* pbegin = myints;                          // ^
    int* pend = myints + sizeof(myints) / sizeof(int); // ^

    pend = std::remove_if(pbegin, pend, IsOdd);     // 2 4 6 8 ? ? ? ? ?
                                                    // ^      ^

    std::cout << "the range contains:";
    for (int* p = pbegin; p != pend; ++p)
        std::cout << ' ' << *p;
    std::cout << '\n';                             // the range contains: 2 4 6 8

    return 0;
}
```

## remove\_copy

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& val);
```

Копіює діапазон, видаляючи значення.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result, за винятком тих елементів, які дорівнюють значенню val. Отриманий діапазон коротший за

[first, last) на стільки елементів, скільки було «видалених» збігів. Функція використовує `operator==` для порівняння окремих елементів із `val`.

Приклад:

```
// remove_copy example
#include <iostream>      // std::cout
#include <algorithm>     // std::remove_copy
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10,20,30,30,20,10,10,20 };
    // 10 20 30 30 20 10 10 20

    std::vector<int> myvector(8);

    std::remove_copy(myints, myints + 8, myvector.begin(), 20);
    // 10 30 30 10 10 0 0 0

    std::cout << "myvector contains: ";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 10 30 30 10 10 0 0 0

    return 0;
}
```

## remove\_copy\_if

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                             OutputIterator result, UnaryPredicate pred);
```

Копіює діапазон, видаляючи значення.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з `result`, за винятком тих елементів, для яких `pred` повертає `true`. Отриманий діапазон коротший за [first, last) на стільки елементів, скільки було «видалених» збігів.

Приклад:

```
// remove_copy_if example
#include <iostream>      // std::cout
#include <algorithm>     // std::remove_copy_if
#include <vector>        // std::vector

bool IsOdd(int i) { return ((i % 2) == 1); }

int main()
{
    int myints[] = { 1,2,3,4,5,6,7,8,9 };
    std::vector<int> myvector(9);

    std::remove_copy_if(myints, myints + 9, myvector.begin(), IsOdd);

    std::cout << "myvector contains: ";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
}
```

```

    // myvector contains: 2 4 6 8 0 0 0 0 0
    return 0;
}

```

## unique

```

// equality (1)
template <class ForwardIterator>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last);

// predicate (2)
template <class ForwardIterator, class BinaryPredicate>
    ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                          BinaryPredicate pred);

```

Видаляє послідовні дублікати в діапазоні.

Видаляє всі елементи, крім першого, з кожної послідовної групи еквівалентних елементів у діапазоні [first, last).

Функція не може змінювати властивості об'єкта, що містить діапазон елементів (тобто, вона не може змінювати розмір масиву або контейнера). Видалення здійснюється заміною елементів-дублікатів на наступний елемент, який не є дублікатором. В якості сигналу про новий розмір скороченого діапазону, функція повертає ітератор до елемента, який слід вважати його елементом після-кінця. Відносний порядок елементів, що не видаляються, зберігається, тоді як елементи між результатом ітератором та last залишаються у дійсному, але невизначеному стані. Функція використовує `operator==` для порівняння пар елементів (або `pred`, у версії (2)).

Приклад:

```

// unique algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::unique, std::distance
#include <vector>        // std::vector

bool myfunction(int i, int j)
{
    return (i == j);
}

int main()
{
    int myints[] = { 10,20,20,20,30,30,20,20,10 };           // 10 20 20 20 30 30 20 20 10
    std::vector<int> myvector(myints, myints + 9);

    // using default comparison:
    std::vector<int>::iterator it;
    it = std::unique(myvector.begin(), myvector.end());      // 10 20 30 20 10 ? ? ? ?
                                                            //                      ^
    myvector.resize(std::distance(myvector.begin(), it));    // 10 20 30 20 10

    // using predicate comparison:
    std::unique(myvector.begin(), myvector.end(), myfunction); // (no changes)
}

```

```

std::cout << "myvector contains: ";
for (it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// myvector contains: 10 20 30 20 10

return 0;
}

```

## unique\_copy

```

// equality (1)
template <class InputIterator, class OutputIterator>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
                              OutputIterator result);

// predicate (2)
template <class InputIterator, class OutputIterator, class BinaryPredicate>
    OutputIterator unique_copy(InputIterator first, InputIterator last,
                              OutputIterator result, BinaryPredicate pred);

```

Копіює діапазон, видаляючи дублікати.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result, за винятком послідовних дублікатів (елементів, які дорівнюють попередньому елементу).

Копіюється лише перший елемент із кожної послідовної групи еквівалентних елементів у діапазоні [first, last). Порівняння елементів виконується або застосуванням `operator==`, або параметра шаблону `pred` (для версії (2)).

Приклад:

```

// unique_copy example
#include <iostream>           // std::cout
#include <algorithm>          // std::unique_copy, std::sort, std::distance
#include <vector>              // std::vector

bool myfunction(int i, int j)
{
    return (i == j);
}

int main()
{
    int myints[] = { 10,20,20,20,30,30,20,20,10 };
    std::vector<int> myvector(9);                                     // 0 0 0 0 0 0 0 0 0

    // using default comparison:
    std::vector<int>::iterator it;
    it = std::unique_copy(myints, myints + 9, myvector.begin());
                                                                    // 10 20 30 20 10 0 0 0 0
                                                                    //                               ^
    std::sort(myvector.begin(), it);
                                                                    // 10 10 20 20 30 0 0 0 0
                                                                    //                               ^

    // using predicate comparison:
    it = std::unique_copy(myvector.begin(), it, myvector.begin(), myfunction);
                                                                    // 10 20 30 20 30 0 0 0 0
                                                                    //                               ^

    myvector.resize(std::distance(myvector.begin(), it)); // 10 20 30
}

```



```

std::cout << "myvector contains: ";
for (it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// myvector contains: 10 20 30

return 0;
}

```

## reverse

```

template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

```

«Перевертає» діапазон.

Змінює порядок елементів у діапазоні [first, last) на зворотний.

Функція викликає iter\_swap для заміни елементів на їх нові місця.

Приклад:

```

// reverse algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::reverse
#include <vector>        // std::vector

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

    std::reverse(myvector.begin(), myvector.end());    // 9 8 7 6 5 4 3 2 1

    std::cout << "myvector contains: ";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 9 8 7 6 5 4 3 2 1

    return 0;
}

```

## reverse\_copy

```

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                           BidirectionalIterator last, OutputIterator result);

```

Копіює діапазон, «перевертаючи» його.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result, але в зворотному порядку.

Приклад:

```

// reverse_copy example
#include <iostream>      // std::cout
#include <algorithm>     // std::reverse_copy
#include <vector>        // std::vector

```

```

int main()
{
    int myints[] = { 1,2,3,4,5,6,7,8,9 };
    std::vector<int> myvector;

    myvector.resize(9);    // allocate space

    std::reverse_copy(myints, myints + 9, myvector.begin());

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 9 8 7 6 5 4 3 2 1

    return 0;
}

```

## rotate

```

template <class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);

```

Циклічний поворот ліворуч елементів діапазону.

Обертає елементи у діапазоні [first, last) таким чином, що елемент, на який вказує middle, стає новим першим елементом.

Приклад:

```

// rotate algorithm example
#include <iostream>    // std::cout
#include <algorithm>    // std::rotate
#include <vector>       // std::vector

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

    std::rotate(myvector.begin(), myvector.begin() + 3, myvector.end());
    // 4 5 6 7 8 9 1 2 3

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 4 5 6 7 8 9 1 2 3

    return 0;
}

```

## rotate\_copy

```

template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);

```

Копіює діапазон з циклічним поворотом ліворуч.

Копіює елементи в діапазоні [first, last) в діапазон, що починається з result, обертаючи елементи таким чином, що елемент, на який вказує middle, стає першим елементом у результуючому діапазоні.

Приклад:

```
// rotate_copy algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::rotate_copy
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10,20,30,40,50,60,70 };

    std::vector<int> myvector(7);

    std::rotate_copy(myints, myints + 3, myints + 7, myvector.begin());

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 40 50 60 70 10 20 30

    return 0;
}
```

## random\_shuffle

```
// generator by default (1)
template <class RandomAccessIterator>
    void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);

// specific generator (2)
template <class RandomAccessIterator, class RandomNumberGenerator>
    void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
        RandomNumberGenerator& gen);
```

Випадково переставляє елементи в діапазоні.

Перевпорядковує елементи в діапазоні [first, last) випадковим чином.

Функція замінює значення кожного елемента зі значенням будь-якого іншого випадково вибраного елемента. При виконанні функція gen визначає, який елемент буде обрано в кожному випадку. В іншому випадку функція використовує деяке невизначене джерело випадковості. Щоб задати рівномірний генератор випадкових випадків, як той, що визначений у <random>, див. shuffle.

Приклад:

```
// random_shuffle example
#include <iostream>      // std::cout
#include <algorithm>     // std::random_shuffle
#include <vector>        // std::vector
#include <ctime>         // std::time
#include <cstdlib>        // std::rand, std::srand
```

```

// random generator function:
int myrandom(int i) { return std::rand() % i; }

int main()
{
    std::srand(unsigned(std::time(0)));
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

    // using built-in random generator:
    std::random_shuffle(myvector.begin(), myvector.end());

    // using myrandom:
    std::random_shuffle(myvector.begin(), myvector.end(), myrandom);

    // print out content:
    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 3 4 1 6 8 9 2 7 5

    return 0;
}

```

## shuffle

```

template <class RandomAccessIterator, class URNG>
void shuffle(RandomAccessIterator first, RandomAccessIterator last, URNG&& g);

```

Випадково переставляє елементи в діапазоні за допомогою генератора.

Переставляє елементи в діапазоні [first, last) випадковим чином, використовуючи g як рівномірний генератор випадкових чисел.

Функція замінює значення кожного елемента зі значенням будь-якого іншого випадково вибраного елемента. Функція визначає елемент, вибраний за допомогою виклику функції g(). Ця функція працює зі стандартними генераторами, як ті, що визначені в <random>. Щоб перетасувати елементи діапазону без такого генератора, див. random\_shuffle.

Приклад:

```

// shuffle algorithm example
#include <iostream> // std::cout
#include <algorithm> // std::shuffle
#include <array> // std::array
#include <random> // std::default_random_engine
#include <chrono> // std::chrono::system_clock

int main()
{
    std::array<int, 5> foo{ 1,2,3,4,5 };

    // obtain a time-based seed:
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();

    shuffle(foo.begin(), foo.end(), std::default_random_engine(seed));
}

```

```
std::cout << "shuffled elements:";
for (int& x : foo) std::cout << ' ' << x;
std::cout << '\n';
// shuffled elements: 3 1 4 2 5

return 0;
}
```

## Розбиття

is_partitioned	Перевірка, чи діапазон – розділений на два. Повертає <b>true</b> , якщо всі елементи в заданому діапазоні, для яких заданий предикат повертає <b>true</b> , передують тим, для яких цей предикат повертає <b>false</b> .
partition	Розділити діапазон на два. Переставляє елементи у заданому діапазоні таким чином, що всі елементи, для яких заданий предикат повертає <b>true</b> , передують всім, для яких цей предикат повертає <b>false</b> .
stable_partition	Розділити діапазон на два із збереженням порядку. Переставляє елементи у заданому діапазоні таким чином, що всі елементи, для яких заданий предикат повертає <b>true</b> , передують всім, для яких цей предикат повертає <b>false</b> , і, на відміну від функції partition, відносний порядок елементів у кожній групі зберігається.
partition_copy	Скопіювати результати розділення діапазону на два. Копіює елементи заданого діапазону, для яких заданий предикат повертає значення <b>true</b> в один діапазон, а ті, для яких цей предикат має значення <b>false</b> – в інший діапазон.
partition_point	Отримати точку розділення. Повертає ітератор першого елемента у заданому розділеному діапазоні, для якого заданий предикат має значення <b>false</b> , вказуючи його точку розділення.

### is\_partitioned

```
template <class InputIterator, class UnaryPredicate>
bool is_partitioned(InputIterator first, InputIterator last, UnaryPredicate pred);
```

Перевіряє, чи діапазон розділений на дві частини.

Повертає **true**, якщо всі елементи в діапазоні [first, last), для яких pred повертає **true**, передують тим, для яких pred повертає **false**.

Якщо діапазон порожній, функція повертає **true**.

Приклад:

```
// is_partitioned example
#include <iostream>      // std::cout
#include <algorithm>     // std::is_partitioned
#include <array>         // std::array

bool IsOdd(int i) { return (i % 2) == 1; }

int main()
{
    std::array<int, 7> foo{ 1,2,3,4,5,6,7 };

    // print contents:
    std::cout << "foo:"; for (int& x : foo) std::cout << ' ' << x;
    if (std::is_partitioned(foo.begin(), foo.end(), IsOdd))
        std::cout << " (partitioned)\n";
    else
        std::cout << " (not partitioned)\n";

    // partition array:
    std::partition(foo.begin(), foo.end(), IsOdd);
```

```

// print contents again:
std::cout << "foo:"; for (int& x : foo) std::cout << ' ' << x;
if (std::is_partitioned(foo.begin(), foo.end(), IsOdd))
    std::cout << " (partitioned)\n";
else
    std::cout << " (not partitioned)\n";
// foo: 1 2 3 4 5 6 7 (not partitioned)
// foo: 1 7 3 5 4 6 2 (partitioned)

return 0;
}

```

## partition

```

template <class BidirectionalIterator, class UnaryPredicate>
    BidirectionalIterator partition(BidirectionalIterator first,
                                   BidirectionalIterator last, UnaryPredicate pred);

```

Розділення діапазону на дві частини.

Переставляє елементи з діапазону `[first, last)` таким чином, що всі елементи, для яких `pred` повертає `true`, передують всім, для яких `pred` повертає `false`. Функція повертає ітератор на перший елемент другої групи.

Відносно впорядкування в межах кожної групи не обов'язково залишиться таким, як було до виклику функції. Див. `stable_partition` для функції з подібною поведінкою, але із збереженням впорядкування у кожній групі.

Приклад:

```

// partition algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::partition
#include <vector>        // std::vector

bool IsOdd(int i) { return (i % 2) == 1; }

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

    std::vector<int>::iterator bound;
    bound = std::partition(myvector.begin(), myvector.end(), IsOdd);

    // print out content:
    std::cout << "odd elements:";
    for (std::vector<int>::iterator it = myvector.begin(); it != bound; ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    std::cout << "even elements:";
    for (std::vector<int>::iterator it = bound; it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // odd elements: 1 9 3 7 5
    // even elements: 6 4 8 2
}

```

```

    return 0;
}

```

## stable\_partition

```

template <class BidirectionalIterator, class UnaryPredicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       UnaryPredicate pred);

```

Розділення діапазону на дві частини із збереженням впорядкування.

Переставляє елементи в діапазоні [first, last) таким чином, що всі елементи, для яких pred повертає true, передують всім, для яких pred повертає false. На відміну від функції partition, відносний порядок елементів у кожній групі зберігається.

Зазвичай це реалізується за допомогою внутрішнього тимчасового буфера.

Приклад:

```

// stable_partition example
#include <iostream>      // std::cout
#include <algorithm>     // std::stable_partition
#include <vector>        // std::vector

bool IsOdd(int i) { return (i % 2) == 1; }

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9

    std::vector<int>::iterator bound;
    bound = std::stable_partition(myvector.begin(), myvector.end(), IsOdd);

    // print out content:
    std::cout << "odd elements:";
    for (std::vector<int>::iterator it = myvector.begin(); it != bound; ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    std::cout << "even elements:";
    for (std::vector<int>::iterator it = bound; it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // odd elements: 1 3 5 7 9
    // even elements: 2 4 6 8

    return 0;
}

```

## partition\_copy

```

template <class InputIterator, class OutputIterator1,
         class OutputIterator2, class UnaryPredicate>
std::pair<OutputIterator1, OutputIterator2>
partition_copy(InputIterator first, InputIterator last,
               OutputIterator1 result_true, OutputIterator2 result_false,
               UnaryPredicate pred);

```



Копіювання з розділенням діапазону на дві частини.

Копіює елементи в діапазоні `[first, last)`, для яких `pred` повертає значення `true` в діапазон, вказаний `result_true`, а ті, для яких `pred` повертає значення `false` – в діапазон, вказаний `result_false`.

Функція повертає `pair` – пару ітераторів на кінці згенерованих послідовностей, вказаних `result_true` і `result_false`, відповідно.

Приклад:

```
// partition_copy example
#include <iostream>      // std::cout
#include <algorithm>     // std::partition_copy, std::count_if
#include <vector>        // std::vector

bool IsOdd(int i) { return (i % 2) == 1; }

int main()
{
    std::vector<int> foo{ 1,2,3,4,5,6,7,8,9 };
    std::vector<int> odd, even;

    // resize vectors to proper size:
    unsigned n = std::count_if(foo.begin(), foo.end(), IsOdd);
    odd.resize(n); even.resize(foo.size() - n);

    // partition:
    std::partition_copy(foo.begin(), foo.end(), odd.begin(), even.begin(), IsOdd);

    // print contents:
    std::cout << "odd: ";
    for (int& x : odd) std::cout << ' ' << x; std::cout << '\n';
    std::cout << "even: ";
    for (int& x : even) std::cout << ' ' << x; std::cout << '\n';
    // odd: 1 3 5 7 9
    // even: 2 4 6 8

    return 0;
}
```

## partition\_point

```
template <class ForwardIterator, class UnaryPredicate>
ForwardIterator partition_point(ForwardIterator first, ForwardIterator last,
                               UnaryPredicate pred);
```

Отримати точку розділення діапазону.

Повертає ітератор до першого елемента в розділеному діапазоні `[first, last)`, для якого `pred` не повертає `true`, тим самим вказуючи точку розділення діапазону.

Елементи в діапазоні вже мають бути розділені, при цьому функція розділення мала бути викликана з тими ж самими аргументами.

Функція оптимізує кількість порівнянь, виконаних шляхом порівняння непослідовних елементів відсортованого діапазону, що є особливо ефективним для ітераторів довільного доступу.

Приклад:

```
// partition_point example
#include <iostream>      // std::cout
#include <algorithm>     // std::partition, std::partition_point
#include <vector>        // std::vector

bool IsOdd(int i) { return (i % 2) == 1; }

int main()
{
    std::vector<int> foo{ 1,2,3,4,5,6,7,8,9 };
    std::vector<int> odd;

    std::partition(foo.begin(), foo.end(), IsOdd);

    auto it = std::partition_point(foo.begin(), foo.end(), IsOdd);
    odd.assign(foo.begin(), it);

    // print contents of odd:
    std::cout << "odd:";
    for (int& x : odd) std::cout << ' ' << x;
    std::cout << '\n';
    // odd: 1 3 5 7 9

    return 0;
}
```

## Сортування

sort	Сортувати елементи в діапазоні
stable_sort	Сортування елементів, що зберігають порядок елементів з еквівалентними значеннями ключа сортування
partial_sort	Частково сортувати елементи в діапазоні. Переставляє елементи заданого діапазону таким чином, що елементи перед заданим значенням стануть найменшими елементами у всьому діапазоні та сортуються за зростанням, тоді як решта елементів залишаються без будь-якого конкретного порядку.
partial_sort_copy	Копіювати та частково сортувати діапазон
is_sorted	Перевірка, чи діапазон – відсортований
is_sorted_until	Знайти перший невідсортований елемент у діапазоні.
nth_element	Сортувати елементи в діапазоні. Переставляє елементи заданого діапазону таким чином, що елемент у n-й позиції є елементом, який знаходився б у цьому положенні у відсортованій послідовності. Інші елементи залишаються без будь-якого конкретного порядку, за винятком того, що жоден з елементів, що передуює n-му, не є більшим за нього, і жоден з елементів, що слідує за ним, не є меншим.

### sort

```
// default (1)
template <class RandomAccessIterator>
    void sort(RandomAccessIterator first, RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Сортувати елементи в діапазоні.

Сортує елементи в діапазоні [first, last) за зростанням. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Не гарантує збереження початкового відносного порядку для еквівалентних елементів (див. `stable_sort`).

Приклад:

```
// sort algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::sort
#include <vector>        // std::vector

bool myfunction(int i, int j) { return (i < j); }

struct myclass
{
    bool operator() (int i, int j) { return (i < j); }
} myobject;

int main()
{
    int myints[] = { 32,71,12,45,26,80,53,33 };
    std::vector<int> myvector(myints, myints + 8);           // 32 71 12 45 26 80 53 33
```

```

// using default comparison (operator <):
std::sort(myvector.begin(), myvector.begin() + 4);      //(12 32 45 71)26 80 53 33

// using function as comp
std::sort(myvector.begin() + 4, myvector.end(), myfunction);
                                                    // 12 32 45 71(26 33 53 80)

// using object as comp
std::sort(myvector.begin(), myvector.end(), myobject);  //(12 26 32 33 45 53 71 80)

std::cout << "myvector contains: ";
for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// myvector contains: 12 26 32 33 45 53 71 80

return 0;
}

```

## stable\_sort

```

// default (1)
template <class RandomAccessIterator>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);

```

Сортувати елементи діапазону, зберігаючи порядок еквівалентних елементів.

Сортує елементи в діапазоні [first, last) за зростанням, як sort, але stable\_sort зберігає відносний порядок елементів з еквівалентними значеннями.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Приклад:

```

// stable_sort example
#include <iostream>      // std::cout
#include <algorithm>     // std::stable_sort
#include <vector>        // std::vector

bool compare_as_ints(double i, double j)
{
    return (int(i) < int(j));
}

int main()
{
    double mydoubles[] = { 3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58 };

    std::vector<double> myvector;

    myvector.assign(mydoubles, mydoubles + 8);

    std::cout << "using default comparison: ";
    std::stable_sort(myvector.begin(), myvector.end());
    for (std::vector<double>::iterator it = myvector.begin();
         it != myvector.end(); ++it)

```

```

        std::cout << ' ' << *it;
std::cout << '\n';

myvector.assign(mydoubles, mydoubles + 8);

std::cout << "using 'compare_as_ints' :";
std::stable_sort(myvector.begin(), myvector.end(), compare_as_ints);
for (std::vector<double>::iterator it = myvector.begin();
      it != myvector.end(); ++it)
    std::cout << ' ' << *it;

std::cout << '\n';
// using default comparison: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
// using 'compare_as_ints' : 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67

return 0;
}

```

## partial\_sort

```

// default (1)
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);

```

Частково сортувати елементи в діапазоні.

Переставляє елементи в діапазоні [first, last) таким чином, що елементи перед middle є найменшими елементами у всьому діапазоні та сортуються за зростанням, тоді як решта елементів залишаються без будь-якого конкретного порядку.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Приклад:

```

// partial_sort example
#include <iostream>      // std::cout
#include <algorithm>     // std::partial_sort
#include <vector>        // std::vector

bool myfunction(int i, int j) { return (i < j); }

int main()
{
    int myints[] = { 9,8,7,6,5,4,3,2,1 };
    std::vector<int> myvector(myints, myints + 9);

    // using default comparison (operator <):
    std::partial_sort(myvector.begin(), myvector.begin() + 5, myvector.end());

    // using function as comp
    std::partial_sort(myvector.begin(), myvector.begin() + 5,
                      myvector.end(), myfunction);

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
}

```

```

std::cout << '\n';
// myvector contains: 1 2 3 4 5 9 8 7 6

return 0;
}

```

## partial\_sort\_copy

```

// default (1)
template <class InputIterator, class RandomAccessIterator>
    RandomAccessIterator
        partial_sort_copy(InputIterator first, InputIterator last,
                          RandomAccessIterator result_first,
                          RandomAccessIterator result_last);

// custom (2)
template <class InputIterator, class RandomAccessIterator, class Compare>
    RandomAccessIterator
        partial_sort_copy(InputIterator first, InputIterator last,
                          RandomAccessIterator result_first,
                          RandomAccessIterator result_last, Compare comp);

```

Копіювати та частково сортувати діапазон.

Копіює найменші елементи в діапазоні [first, last) до [result\_first, result\_last), сортуючи скопійовані елементи. Кількість скопійованих елементів така ж, як і відстань між result\_first та result\_last (якщо це не більше кількості елементів у [first, last)).

Діапазон [first, last) не змінюється. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Приклад:

```

// partial_sort_copy example
#include <iostream>      // std::cout
#include <algorithm>     // std::partial_sort_copy
#include <vector>        // std::vector

bool myfunction(int i, int j) { return (i < j); }

int main()
{
    int myints[] = { 9,8,7,6,5,4,3,2,1 };
    std::vector<int> myvector(5);

    // using default comparison (operator <):
    std::partial_sort_copy(myints, myints + 9, myvector.begin(), myvector.end());

    // using function as comp
    std::partial_sort_copy(myints, myints + 9,
                          myvector.begin(), myvector.end(), myfunction);

    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 1 2 3 4 5

    return 0;
}

```

## is\_sorted

```
// default (1)
template <class ForwardIterator>
    bool is_sorted(ForwardIterator first, ForwardIterator last);

// custom (2)
template <class ForwardIterator, class Compare>
    bool is_sorted(ForwardIterator first, ForwardIterator last, Compare comp);
```

Перевіряє, чи діапазон – відсортований. Повертає `true`, якщо діапазон `[first, last)` відсортований за зростанням. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Приклад:

```
// is_sorted example
#include <iostream>          // std::cout
#include <algorithm>         // std::is_sorted, std::prev_permutation
#include <array>             // std::array

int main()
{
    std::array<int, 4> foo{ 2,4,1,3 };

    do {
        // try a new permutation:
        std::prev_permutation(foo.begin(), foo.end());

        // print range:
        std::cout << "foo:";
        for (int& x : foo) std::cout << ' ' << x;
        std::cout << '\n';

    } while (!std::is_sorted(foo.begin(), foo.end()));

    std::cout << "the range is sorted!\n";
    // foo: 2 3 4 1
    // foo: 2 3 1 4
    // foo: 2 1 4 3
    // foo: 2 1 3 4
    // foo: 1 4 3 2
    // foo: 1 4 2 3
    // foo: 1 3 4 2
    // foo: 1 3 2 4
    // foo: 1 2 4 3
    // foo: 1 2 3 4
    // the range is sorted!
}
```

## is\_sorted\_until

```
// default (1)
template <class ForwardIterator>
    ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

// custom (2)
template <class ForwardIterator, class Compare>
    ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
                                    Compare comp);
```

Знайти перший несортований елемент в діапазоні.

Повертає ітератор до першого елемента в діапазоні `[first, last)`, який порушує порядок зростання. Діапазон між `first` і результируючим ітератором – відсортований за зростанням. Якщо відсортовано весь діапазон, функція повертає `last`.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Приклад:

```
// is_sorted_until example
#include <iostream>      // std::cout
#include <algorithm>     // std::is_sorted_until, std::prev_permutation
#include <array>         // std::array

int main()
{
    std::array<int, 4> foo{ 2,4,1,3 };
    std::array<int, 4>::iterator it;

    do {
        // try a new permutation:
        std::prev_permutation(foo.begin(), foo.end());

        // print range:
        std::cout << "foo:";
        for (int& x : foo) std::cout << ' ' << x;
        it = std::is_sorted_until(foo.begin(), foo.end());
        std::cout << " (" << (it - foo.begin()) << " elements sorted)\n";

    } while (it != foo.end());

    std::cout << "the range is sorted!\n";
    // foo: 2 3 4 1 (3 elements sorted)
    // foo: 2 3 1 4 (2 elements sorted)
    // foo: 2 1 4 3 (1 elements sorted)
    // foo: 2 1 3 4 (1 elements sorted)
    // foo: 1 4 3 2 (2 elements sorted)
    // foo: 1 4 2 3 (2 elements sorted)
    // foo: 1 3 4 2 (3 elements sorted)
    // foo: 1 3 2 4 (2 elements sorted)
    // foo: 1 2 4 3 (3 elements sorted)
    // foo: 1 2 3 4 (4 elements sorted)
    // the range is sorted!

    return 0;
}
```

## **nth\_element**

```
// default (1)
template <class RandomAccessIterator>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                    RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                    RandomAccessIterator last, Compare comp);
```

Сортувати елементи в діапазоні.



Переставляє елементи в діапазоні `[first, last)` таким чином, що елемент у  $n$ -й позиції є елементом, який буде у цій позиції у відсортованій послідовності. Інші елементи залишаються без будь-якого конкретного порядку, за винятком того, що жоден з елементів, що передуює  $n$ -му, не є більшим за нього, і жоден з елементів, що слідує за ним, не є меншим.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Приклад:

```
// nth_element example
#include <iostream>      // std::cout
#include <algorithm>     // std::nth_element, std::random_shuffle
#include <vector>        // std::vector

bool myfunction(int i, int j) { return (i < j); }

int main()
{
    std::vector<int> myvector;

    // set some values:
    for (int i = 1; i < 10; i++) myvector.push_back(i);    // 1 2 3 4 5 6 7 8 9

    std::random_shuffle(myvector.begin(), myvector.end());

    // using default comparison (operator <):
    std::nth_element(myvector.begin(), myvector.begin() + 5, myvector.end());

    // using function as comp
    std::nth_element(myvector.begin(), myvector.begin() + 5,
                     myvector.end(), myfunction);

    // print out content:
    std::cout << "myvector contains:";
    for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // myvector contains: 3 1 4 2 5 6 9 7 8

    return 0;
}
```

## Бінарний пошук

(працює на розділених / відсортованих діапазонах)

lower_bound	Повернути ітератор до нижньої межі
upper_bound	Повернути ітератор до верхньої межі
equal_range	Отримати піддіапазон рівних елементів
binary_search	Перевірка, чи існує значення у відсортованій послідовності

### lower\_bound

```
// default (1)
template <class ForwardIterator, class T>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                               const T & val);

// custom (2)
template <class ForwardIterator, class T, class Compare>
    ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                               const T & val, Compare comp);
```

Повертає ітератор до першого елемента, який не менший заданого значення.

Повертає ітератор, що вказує на перший елемент у діапазоні [first, last), який не менший, ніж val. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Елементи в діапазоні вже мають бути відсортовані за цим самим критерієм (`operator<` або `comp`) або, принаймні, розділені по відношенню до val.

Функція оптимізує кількість порівнянь, виконаних шляхом порівняння непослідовних елементів відсортованого діапазону, що є особливо ефективним для ітераторів довільного доступу. На відміну від `upper_bound`, значення, вказане ітератором, який повертає ця функція, може бути не лише більшим за val, а і дорівнювати йому.

Див. приклад до `upper_bound`.

### upper\_bound

```
// default (1)
template <class ForwardIterator, class T>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                               const T & val);

// custom (2)
template <class ForwardIterator, class T, class Compare>
    ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                               const T & val, Compare comp);
```

Повертає ітератор до першого елемента, що перевищує певне значення.

Повертає ітератор, що вказує на перший елемент в діапазоні [first, last), який більший, ніж val. Елементи порівнюються за допомогою `operator<` для першої версії та `comp`

для другої. Елементи в діапазоні вже мають бути відсортовані за цим самим критерієм (`operator<` або `comp`) або, принаймні, розділені по відношенню до `val`.

Функція оптимізує кількість порівнянь, виконаних шляхом порівняння непослідовних елементів відсортованого діапазону, що є особливо ефективним для ітераторів довільного доступу. На відміну від `lower_bound`, значення, вказане ітератором, який повертає ця функція, не може бути еквівалентним `val`, лише більшим.

Приклад:

```
// lower_bound / upper_bound example
#include <iostream>      // std::cout
#include <algorithm>     // std::lower_bound, std::upper_bound, std::sort
#include <vector>        // std::vector

int main()
{
    int myints[] = { 10,20,30,30,20,10,10,20 };
    std::vector<int> v(myints, myints + 8);           // 10 20 30 30 20 10 10 20

    std::sort(v.begin(), v.end());                    // 10 10 10 20 20 20 30 30

    std::vector<int>::iterator low, up;
    low = std::lower_bound(v.begin(), v.end(), 20); //           ^
    up = std::upper_bound(v.begin(), v.end(), 20);  //           ^

    std::cout << "lower_bound at position " << (low - v.begin()) << '\n';
    std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
    // lower_bound at position 3
    // upper_bound at position 6

    return 0;
}
```

## **equal\_range**

```
// default (1)
template <class ForwardIterator, class T>
    std::pair<ForwardIterator, ForwardIterator>
        equal_range(ForwardIterator first, ForwardIterator last, const T & val);

// custom (2)
template <class ForwardIterator, class T, class Compare>
    std::pair<ForwardIterator, ForwardIterator>
        equal_range(ForwardIterator first, ForwardIterator last, const T & val,
                    Compare comp);
```

Отримати піддіапазон елементів, які дорівнюють заданому значенню.

Повертає межі піддіапазону, що включає всі елементи діапазону `[first, last)` зі значеннями, еквівалентними `val`. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, `a` і `b`, вважаються еквівалентними, якщо `(!(a<b) && !(b<a))` або якщо `(!comp(a, b) && !comp(b, a))`.

Елементи в діапазоні вже мають бути відсортовані за цим самим критерієм (`operator<` або `comp`) або, принаймні, розділені по відношенню до `val`. Якщо `val` не еквівалентно будь-

якому значенню в діапазоні, результуючий піддіапазон має нульову довжину, причому обидва ітератори вказують на найближче значення, більше ніж val, якщо воно є, або на останнє, якщо val більше, ніж усі елементи в діапазоні.

Приклад:

```
// equal_range example
#include <iostream>      // std::cout
#include <algorithm>     // std::equal_range, std::sort
#include <vector>        // std::vector

bool mygreater(int i, int j) { return (i > j); }

int main()
{
    int myints[] = { 10,20,30,30,20,10,10,20 };
    std::vector<int> v(myints, myints + 8);           // 10 20 30 30 20 10 10 20

    std::pair<std::vector<int>::iterator, std::vector<int>::iterator> bounds;

    // using default comparison:
    std::sort(v.begin(), v.end());                    // 10 10 10 20 20 20 30 30
    bounds = std::equal_range(v.begin(), v.end(), 20); //           ^         ^

    // using "mygreater" as comp:
    std::sort(v.begin(), v.end(), mygreater);          // 30 30 20 20 20 10 10 10
    bounds =                                           //           ^         ^
        std::equal_range(v.begin(), v.end(), 20, mygreater);

    std::cout << "bounds at positions " << (bounds.first - v.begin());
    std::cout << " and " << (bounds.second - v.begin()) << '\n';
    // bounds at positions 2 and 5

    return 0;
}
```

## binary\_search

```
// default (1)
template <class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T & val);

// custom (2)
template <class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T & val, Compare comp);
```

Перевірка, чи існує шукане значення у відсортованій послідовності.

Повертає true, якщо у діапазоні [first, last) є елемент, еквівалентний val, та false – в іншому випадку.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, a і b, вважаються еквівалентними, якщо `!(a<b) && !(b<a)` або якщо `!comp(a, b) && !comp(b, a)`.

Елементи в діапазоні вже мають бути відсортовані за цим самим критерієм (`operator<` або `comp`) або, принаймні, розділені по відношенню до `val`.

Функція оптимізує кількість порівнянь, виконаних шляхом порівняння непослідовних елементів відсортованого діапазону, що є особливо ефективним для ітераторів довільного доступу.

Приклад:

```
// binary_search example
#include <iostream>      // std::cout
#include <algorithm>     // std::binary_search, std::sort
#include <vector>        // std::vector

bool myfunction(int i, int j) { return (i < j); }

int main()
{
    int myints[] = { 1,2,3,4,5,4,3,2,1 };
    std::vector<int> v(myints, myints + 9); // 1 2 3 4 5 4 3 2 1

    // using default comparison:
    std::sort(v.begin(), v.end());

    std::cout << "looking for a 3... ";
    if (std::binary_search(v.begin(), v.end(), 3))
        std::cout << "found!\n";
    else
        std::cout << "not found.\n";

    // using myfunction as comp:
    std::sort(v.begin(), v.end(), myfunction);

    std::cout << "looking for a 6... ";
    if (std::binary_search(v.begin(), v.end(), 6, myfunction))
        std::cout << "found!\n";
    else
        std::cout << "not found.\n";
    // looking for a 3... found!
    // looking for a 6... not found.

    return 0;
}
```

## Злиття

(працює на відсортованих діапазонах)

merge	Об'єднати відсортовані діапазони
inplace_merge	Об'єднати послідовні відсортовані діапазони
includes	Перевірка, чи містить відсортований діапазон інший відсортований діапазон
set_union	Об'єднання двох відсортованих діапазонів
set_intersection	Перетин двох відсортованих діапазонів
set_difference	Різниця двох відсортованих діапазонів
set_symmetric_difference	Симетрична різниця двох відсортованих діапазонів

### merge

```
// default (1)
template <class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);

// custom (2)
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
    OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);
```

Злиття відсортованих діапазонів.

Поеднує елементи в відсортованих діапазонах [first1, last1) і [first2, last2), в новий діапазон, що починається з result, усі елементи якого – відсортовані.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Елементи в обох діапазонах вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`). Отриманий діапазон також сортується відповідно до цього критерію.

Приклад:

```
// merge algorithm example
#include <iostream>          // std::cout
#include <algorithm>         // std::merge, std::sort
#include <vector>            // std::vector

int main()
{
    int first[] = { 5,10,15,20,25 };
    int second[] = { 50,40,30,20,10 };
    std::vector<int> v(10);

    std::sort(first, first + 5);
    std::sort(second, second + 5);
    std::merge(first, first + 5, second, second + 5, v.begin());

    std::cout << "The resulting vector contains:";
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
```

```

std::cout << '\n';
// The resulting vector contains: 5 10 10 15 20 20 25 30 40 50

return 0;
}

```

## inplace\_merge

```

// default (1)
template <class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
                  BidirectionalIterator last);

// custom (2)
template <class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
                  BidirectionalIterator last, Compare comp);

```

Злиття послідовних відсортованих діапазонів.

Об'єднує два послідовні відсортовані діапазони: [first, middle) та [middle, last), розміщуючи результат у комбінованому відсортованому діапазоні [first, last).

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Елементи в обох діапазонах вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`). Отриманий діапазон також сортується відповідно до цього критерію.

Функція зберігає відносний порядок елементів з еквівалентними значеннями, причому елементи з першого діапазону передують їх еквівалентам з другого.

Приклад:

```

// inplace_merge example
#include <iostream>      // std::cout
#include <algorithm>     // std::inplace_merge, std::sort, std::copy
#include <vector>        // std::vector

int main()
{
    int first[] = { 5,10,15,20,25 };
    int second[] = { 50,40,30,20,10 };
    std::vector<int> v(10);
    std::vector<int>::iterator it;

    std::sort(first, first + 5);
    std::sort(second, second + 5);

    it = std::copy(first, first + 5, v.begin());
    std::copy(second, second + 5, it);

    std::inplace_merge(v.begin(), v.begin() + 5, v.end());

    std::cout << "The resulting vector contains:";
    for (it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // The resulting vector contains: 5 10 10 15 20 20 25 30 40 50

    return 0;
}

```

## includes

```
// default (1)
template <class InputIterator1, class InputIterator2>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2);

// custom (2)
template <class InputIterator1, class InputIterator2, class Compare>
    bool includes(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Перевірка, чи включає відсортований діапазон інший відсортований діапазон.

Повертає true, якщо відсортований діапазон [first1, last1) містить усі елементи відсортованого діапазону [first2, last2).

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої.

Два елементи, `a` і `b`, вважаються еквівалентними, якщо  $(!(a < b) \ \&\& \ !(b < a))$  або якщо  $(!comp(a, b) \ \&\& \ !comp(b, a))$ .

Елементи в діапазоні вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`).

Приклад:

```
// includes algorithm example
#include <iostream>      // std::cout
#include <algorithm>     // std::includes, std::sort

bool myfunction(int i, int j) { return i < j; }

int main()
{
    int container[] = { 5,10,15,20,25,30,35,40,45,50 };
    int continent[] = { 40,30,20,10 };

    std::sort(container, container + 10);
    std::sort(continent, continent + 4);

    // using default comparison:
    if (std::includes(container, container + 10, continent, continent + 4))
        std::cout << "container includes continent!\n";

    // using myfunction as comp:
    if (std::includes(container, container + 10, continent, continent + 4, myfunction))
        std::cout << "container includes continent!\n";
    // container includes continent!
    // container includes continent!

    return 0;
}
```

## set\_union

```
// default (1)
template <class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
```



```
// custom (2)
template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);
```

Об'єднання двох відсортованих діапазонів.

Будує відсортований діапазон, розташований починаючи з позиції, вказаній `result`, на основі об'єднання як множин двох відсортованих діапазонів `[first1, last1)` і `[first2, last2)`. Об'єднання двох множин утворюється елементами, які присутні або в одному з наборів, або в обох. Елементи з другого діапазону, які мають еквівалентний елемент у першому діапазоні, не копіюються в результуючий діапазон. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, `a` і `b`, вважаються еквівалентними, якщо `(!(a < b) && !(b < a))` або якщо `(!comp(a, b) && !comp(b, a))`. Елементи в діапазонах вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`). Отриманий діапазон також сортується відповідно до цього критерію.

Приклад:

```
// set_union example
#include <iostream>      // std::cout
#include <algorithm>     // std::set_union, std::sort
#include <vector>        // std::vector

int main()
{
    int first[] = { 5,10,15,20,25 };
    int second[] = { 50,40,30,20,10 };
    std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
    std::vector<int>::iterator it;

    std::sort(first, first + 5);      // 5 10 15 20 25
    std::sort(second, second + 5);    // 10 20 30 40 50

    it = std::set_union(first, first + 5, second, second + 5, v.begin());
    // 5 10 15 20 25 30 40 50 0 0
    v.resize(it - v.begin());         // 5 10 15 20 25 30 40 50

    std::cout << "The union has " << (v.size()) << " elements:\n";
    for (it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // The union has 8 elements:
    // 5 10 15 20 25 30 40 50

    return 0;
}
```

## set\_intersection

```
// default (1)
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result);
```

```
// custom (2)
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator result, Compare comp);
```

Перетин двох відсортованих діапазонів.

Будує відсортований діапазон, розташований починаючи з позиції, вказаній `result`, на основі перетину як множин двох відсортованих діапазонів `[first1, last1)` і `[first2, last2)`.

Перетин двох множин утворюють лише елементи, присутні в обох множинах. Елементи, скопійовані функцією, беруться завжди з першого діапазону в тому ж самому порядку. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, `a` і `b`, вважаються еквівалентними, якщо `!(a<b) && !(b<a)` або якщо `(!comp(a, b) && !comp(b, a))`.

Елементи в діапазонах вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`). Отриманий діапазон також сортується відповідно до цього критерію.

Приклад:

```
// set_intersection example
#include <iostream>      // std::cout
#include <algorithm>     // std::set_intersection, std::sort
#include <vector>        // std::vector

int main()
{
    int first[] = { 5,10,15,20,25 };
    int second[] = { 50,40,30,20,10 };
    std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
    std::vector<int>::iterator it;

    std::sort(first, first + 5);      // 5 10 15 20 25
    std::sort(second, second + 5);    // 10 20 30 40 50

    it = std::set_intersection(first, first + 5, second, second + 5, v.begin());
    // 10 20 0 0 0 0 0 0 0 0
    v.resize(it - v.begin());         // 10 20

    std::cout << "The intersection has " << (v.size()) << " elements:\n";
    for (it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // The intersection has 2 elements:
    // 10 20

    return 0;
}
```

## set\_difference

```
//default (1)
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
```

```
// custom (2)
template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
```

Різниця двох відсортованих діапазонів.

Будує відсортований діапазон, розташований починаючи з позиції, вказаній `result`, на основі різниці як множин двох відсортованих діапазонів `[first1, last1)` і `[first2, last2)`.

Різниця двох множин формується елементами, які є в першій, але не в другій множині. Елементи, скопійовані функцією, беруться завжди з першого діапазону в тому ж самому порядку. Для контейнерів, що підтримують багатократне входження значення, різниця включає стільки входжень цього значення, скільки є в першому діапазоні, мінус кількість відповідних елементів у другому, при цьому зберігається порядок розташування елементів. Це – несиметрична операція, для симетричного еквівалента див. `set_symmetric_difference`.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, `a` і `b`, вважаються еквівалентними, якщо `!(a<b) && !(b<a)` або якщо `!comp(a, b) && !comp(b, a)`.

Елементи в діапазонах вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`). Отриманий діапазон також сортується відповідно до цього критерію.

Приклад:

```
// set_difference example
#include <iostream>      // std::cout
#include <algorithm>     // std::set_difference, std::sort
#include <vector>        // std::vector

int main()
{
    int first[] = { 5,10,15,20,25 };
    int second[] = { 50,40,30,20,10 };
    std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
    std::vector<int>::iterator it;

    std::sort(first, first + 5);     // 5 10 15 20 25
    std::sort(second, second + 5);   // 10 20 30 40 50

    it = std::set_difference(first, first + 5, second, second + 5, v.begin());
    // 5 15 25 0 0 0 0 0 0 0
    v.resize(it - v.begin());        // 5 15 25

    std::cout << "The difference has " << (v.size()) << " elements:\n";
    for (it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';              // The difference has 3 elements:
    // 5 15 25

    return 0;
}
```

## set\_symmetric\_difference

```
// default (1)
template <class InputIterator1, class InputIterator2, class OutputIterator>
    OutputIterator set_symmetric_difference(InputIterator1 first1,
                                           InputIterator1 last1,
                                           InputIterator2 first2,
                                           InputIterator2 last2,
                                           OutputIterator result);

// custom (2)
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
    OutputIterator set_symmetric_difference(InputIterator1 first1,
                                           InputIterator1 last1,
                                           InputIterator2 first2,
                                           InputIterator2 last2,
                                           OutputIterator result, Compare comp);
```

Симетрична різниця двох відсортованих діапазонів.

Будує відсортований діапазон, розташований починаючи з позиції, вказаній result, на основі симетричної різниці як множин двох відсортованих діапазонів [first1, last1) і [first2, last2).

Симетрична різниця двох множин формується елементами, які присутні в одній із множин, але не в іншій. Перед викликом функції відкидаються дублікати серед еквівалентних елементів у кожному діапазоні. Існуючий порядок зберігається і для скопійованих елементів. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, `a` і `b`, вважаються еквівалентними, якщо `!(a<b) && !(b<a)` або якщо `!comp(a, b) && !comp(b, a)`.

Елементи в діапазонах вже мають бути впорядковані за цим самим критерієм (`operator<` або `comp`). Отриманий діапазон також сортується відповідно до цього критерію.

Приклад:

```
// set_symmetric_difference example
#include <iostream>      // std::cout
#include <algorithm>     // std::set_symmetric_difference, std::sort
#include <vector>        // std::vector

int main()
{
    int first[] = { 5,10,15,20,25 };
    int second[] = { 50,40,30,20,10 };
    std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
    std::vector<int>::iterator it;

    std::sort(first, first + 5);      // 5 10 15 20 25
    std::sort(second, second + 5);    // 10 20 30 40 50

    it = std::set_symmetric_difference(first, first + 5,
                                       second, second + 5, v.begin());
                                       // 5 15 25 30 40 50 0 0 0 0
    v.resize(it - v.begin());         // 5 15 25 30 40 50
```

```

std::cout << "The symmetric difference has " << (v.size()) << " elements:\n";
for (it = v.begin(); it != v.end(); ++it)
    std::cout << ' ' << *it;
std::cout << '\n';
// The symmetric difference has 6 elements:
//  5 15 25 30 40 50

return 0;
}

```

## Максимальна купа

push_heap	Вставити елемент у максимальну купу
pop_heap	Вилучити елемент із максимальної купи
make_heap	Зробити максимальну купу з діапазону
sort_heap	Сортування елементів в максимальній купі
is_heap	Перевірка, чи є діапазон максимальною купою
is_heap_until	Знайти перший елемент не в порядку максимальної купи

Максимальна купа – це набір елементів в діапазоні  $[first, last)$ , що має такі властивості:

- при  $N = last - first$ , для всіх  $0 < i < N$ , елемент  $*(first + \text{floor}((i-1)/2.0))$  не менший, ніж елемент  $*(first + i)$ . Функція  $\text{floor}(x)$  повертає найбільше ціле число (представлене як `double`), яке не більше, ніж  $x$ .
- новий елемент можна додати за допомогою `std::push_heap()`.
- перший елемент можна видалити за допомогою `std::pop_heap()`.

Максимальна купа – це спосіб впорядкування елементів діапазону, що дозволяє швидко отримувати елемент з найбільшим значенням у будь-який момент (за допомогою `pop_heap`), одночасно дозволяючи швидко вставляти нові елементи (за допомогою `push_heap`).

Елемент з найбільшим значенням завжди розташований першим. Порядок інших елементів залежить від конкретної реалізації, але він узгоджується з усіма функціями, пов'язаними з максимальною купою.

Стандартний адаптер контейнера `priority_queue` автоматично викликає `make_heap`, `push_heap` і `pop_heap` для підтримки властивостей максимальної купи для цього контейнера.

### push\_heap

```
// default (1)
template <class RandomAccessIterator>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    void push_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

Вставка елемента у діапазон максимальної купи.

Розглядаючи максимальну купу в діапазоні  $[first, last-1)$ , ця функція розширює діапазон, що вважається максимальною купою, до  $[first, last)$ , розміщуючи значення в позиції  $(last-1)$  у відповідну позицію всередині діапазону  $[first, last)$ .

Діапазон можна перетворити в максимальну купу, викликавши `make_heap`. Після цього властивості максимальної купи для такого діапазону зберігаються, якщо елементи додаються та видаляються з нього за допомогою `push_heap` та `pop_heap`, відповідно.

Див. приклад до `sort_heap`.

### **pop\_heap**

```
// default (1)
template <class RandomAccessIterator>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
```

Вилучення елемента з діапазону максимальної купи.

Переставляє елементи в діапазоні `[first, last)` таким чином, що частина, що вважається максимальною купою, скорочується на один елемент. Елемент з найбільшим значенням переміщується до позиції `(last-1)`.

Поки елемент з найбільшим значенням переміщується з позиції `first` в позицію `(last-1)` (яка тепер буде поза максимальною купою), інші елементи реорганізуються таким чином, що діапазон `[first, last-1)` зберігає властивості максимальної купи.

Діапазон можна перетворити в максимальну купу, викликавши `make_heap`. Після цього властивості максимальної купи для такого діапазону зберігаються, якщо елементи додаються та видаляються з нього за допомогою `push_heap` та `pop_heap`, відповідно.

Див. приклад до `sort_heap`.

### **make\_heap**

```
// default (1)
template <class RandomAccessIterator>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    void make_heap(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
```

Зробити максимальну купу з діапазону.

Переставляє елементи в діапазоні `[first, last)` таким чином, що вони утворюють максимальну купу. Максимальна купа – це спосіб впорядкувати елементи діапазону, що дозволяє швидко отримувати елемент з найбільшим значенням у будь-який момент (за допомогою `pop_heap`), навіть неодноразово, одночасно дозволяючи швидко вставляти нові елементи (за допомогою `push_heap`).

Елемент з найбільшим значенням завжди вказується першим. Порядок інших елементів залежить від конкретної реалізації, але він узгоджується з усіма функціями, пов'язаними з максимальною купою. Елементи порівнюються за допомогою `operator<` (для першої версії) або `comp` (для другої). Елемент з найвищим значенням – це елемент, для якого результатом буде `false` при порівнянні з кожним іншим елементом у діапазоні.

Стандартний адаптер контейнера `priority_queue` автоматично викликає `make_heap`, `push_heap` та `pop_heap` для підтримки властивостей максимальної купи для цього контейнера.

Див. приклад до `sort_heap`.

## `sort_heap`

```
// default (1)
template <class RandomAccessIterator>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

// custom(2)
template <class RandomAccessIterator, class Compare>
    void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

Сортувати елементи максимальної купи.

Сортує елементи в діапазоні максимальної купи `[first, last)` за зростанням. Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої, цей критерій має бути таким самим, як і той, що використовувався для побудови купи. Після виконання функції діапазон `[first, last)` втрачає свої властивості максимальної купи.

Приклад:

```
// range heap example
#include <iostream>    // std::cout
#include <algorithm>   // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
#include <vector>      // std::vector

void print(const char* text, std::vector<int> v)
{
    std::cout << text << ":" << '\n';
    for (const auto& e : v)
        std::cout << ' ' << e;
    std::cout << "\n\n";
}

int main()
{
    int myints[] = { 10,20,30,5,15 };
    std::vector<int> v(myints, myints + 5);
    print("initial vector", v);

    std::make_heap(v.begin(), v.end());
    print("initial max heap", v);
    std::cout << "front in max heap          : " << v.front() << "\n\n";

    std::pop_heap(v.begin(), v.end()); v.pop_back();
    print("max heap after pop", v);
}
```



```

std::cout << "front in max heap after pop : " << v.front() << "\n\n";

v.push_back(99); std::push_heap(v.begin(), v.end());
print("max heap after push", v);
std::cout << "front in max heap after push: " << v.front() << "\n\n";

std::sort_heap(v.begin(), v.end());
print("max heap after sort", v);

return 0;
}

```

```

Select Microsoft Visual Studio De...
initial vector:
10 20 30 5 15

initial max heap:
30 20 10 5 15

front in max heap          : 30

max heap after pop:
20 15 10 5

front in max heap after pop : 20

max heap after push:
99 20 10 5 15

front in max heap after push: 99

max heap after sort:
5 10 15 20 99

C:\Users\Biktop\source\repos\Projec

```

## is\_heap

```

// default (1)
template <class RandomAccessIterator>
    bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
        Compare comp);

```

Перевірка, чи діапазон є максимальною купою

Повертає true, якщо діапазон [first, last) утворює максиманьну купу, яка була би побудована за допомогою make\_heap.

Елементи порівнюються за допомогою operator< для першої версії та comp для другої.

Приклад:

```

// is_heap example
#include <iostream>      // std::cout
#include <algorithm>     // std::is_heap, std::make_heap, std::pop_heap
#include <vector>        // std::vector

int main()
{
    std::vector<int> foo{ 9,5,2,6,4,1,3,8,7 };

    if (!std::is_heap(foo.begin(), foo.end()))
        std::make_heap(foo.begin(), foo.end());
}

```

```

std::cout << "Popping out elements:";
while (!foo.empty()) {
    std::pop_heap(foo.begin(), foo.end()); // moves largest element to back
    std::cout << ' ' << foo.back();       // prints back
    foo.pop_back();                       // pops element out of container
}
std::cout << '\n';
// Popping out elements: 9 8 7 6 5 4 3 2 1

return 0;
}

```

## is\_heap\_until

```

// default (1)
template <class RandomAccessIterator>
    RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                       RandomAccessIterator last);

// custom (2)
template <class RandomAccessIterator, class Compare>
    RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                       RandomAccessIterator last,
                                       Compare comp);

```

Знайти перший елемент не в порядку максимальної купи.

Повертає ітератор до першого елемента в діапазоні [first, last), який не розташований у правильній позиції, якщо діапазон вважається максимальною купою (яка була би побудована за допомогою make\_heap).

Діапазон між first і результатом ітератором — це максимальна купа.

Якщо весь діапазон є допустимою купою, функція повертається last.

Елементи порівнюються за допомогою operator< для першої версії та comp для другої.

Приклад:

```

// is_heap example
#include <iostream>      // std::cout
#include <algorithm>     // std::is_heap_until, std::sort, std::reverse
#include <vector>        // std::vector

int main()
{
    std::vector<int> foo{ 2,6,9,3,8,4,5,1,7 };

    std::sort(foo.begin(), foo.end());
    std::reverse(foo.begin(), foo.end());

    auto last = std::is_heap_until(foo.begin(), foo.end());

    std::cout << "The " << (last - foo.begin()) << " first elements are a valid heap:";
    for (auto it = foo.begin(); it != last; ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';
    // The 9 first elements are a valid heap: 9 8 7 6 5 4 3 2 1

    return 0;
}

```

## Min / max

min	Повернути найменше з двох
max	Повернути найбільше з двох
minmax	Повернути найменший і найбільший елементи з двох
min_element	Повернути найменший елемент у діапазоні
max_element	Повернути найбільший елемент у діапазоні
minmax_element	Повернути найменший і найбільший елементи в діапазоні

### min

```
// default (1)
template <class T>
    const T& min(const T& a, const T& b);

// custom (2)
template <class T, class Compare>
    const T& min(const T& a, const T& b, Compare comp);
```

Найменше з двох значень.

Повертає найменше значення з а та b. Якщо обидва еквівалентні, повертається а.

Функція використовує `operator<` (або `comp`, у версії (2)) для порівняння значень.

Приклад:

```
// min example
#include <iostream>      // std::cout
#include <algorithm>     // std::min

int main()
{
    std::cout << "min(1, 2) == " << std::min(1, 2) << '\n';
    std::cout << "min(2, 1) == " << std::min(2, 1) << '\n';
    std::cout << "min('a', 'z') == " << std::min('a', 'z') << '\n';
    std::cout << "min(3.14, 2.72) == " << std::min(3.14, 2.72) << '\n';
    // min(1, 2) == 1
    // min(2, 1) == 1
    // min('a', 'z') == a
    // min(3.14, 2.72) == 2.72

    return 0;
}
```

### max

```
// default (1)
template <class T>
    const T& max(const T& a, const T& b);

// custom (2)
template <class T, class Compare>
    const T& max(const T& a, const T& b, Compare comp);
```

Найбільше з двох значень.

Повертає найбільше значення з а та b. Якщо обидва еквівалентні, повертається а.

Функція використовує `operator<` (або `comp`, для версії (2)) для порівняння значень.

Приклад:

```
// max example
#include <iostream>      // std::cout
#include <algorithm>     // std::max

int main()
{
    std::cout << "max(1, 2) == " << std::max(1, 2) << '\n';
    std::cout << "max(2, 1) == " << std::max(2, 1) << '\n';
    std::cout << "max('a', 'z') == " << std::max('a', 'z') << '\n';
    std::cout << "max(3.14, 2.73) == " << std::max(3.14, 2.73) << '\n';
    // max(1, 2) == 2
    // max(2, 1) == 2
    // max('a', 'z') == z
    // max(3.14, 2.73) == 3.14

    return 0;
}
```

## minmax

```
// default (1)
template <class T>
    std::pair<const T&, const T&> minmax(const T& a, const T& b);

// custom (2)
template <class T, class Compare>
    std::pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

Пара найменшого і найбільшого з двох значень.

Повертає пару з найменшим із `a` і `b` як першим елементом, та найбільшим – як другим.

Якщо обидва еквівалентні, функція повертає `make_pair (a, b)`.

Функція використовує `оператор<` (або `comp`, для версії (2)) для порівняння значень.

Приклад:

```
// minmax example
#include <iostream>      // std::cout
#include <algorithm>     // std::minmax

int main()
{
    auto result = std::minmax({ 1,2,3,4,5 });

    std::cout << "minmax({1,2,3,4,5}): ";
    std::cout << result.first << ' ' << result.second << '\n';
    // minmax({1,2,3,4,5}): 1 5

    return 0;
}
```

## min\_element

```
// default (1)
template <class ForwardIterator>
    ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

```
// custom (2)
template <class ForwardIterator, class Compare>
    ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                               Compare comp);
```

Найменший елемент в діапазоні.

Повертає ітератор, що вказує на елемент з найменшим значенням в діапазоні [first, last).

Порівняння виконуються за допомогою або `operator<` для першої версії, або `comp` для другої. Елемент є найменшим, якщо немає інших елементів, менших за нього. Якщо цій умові відповідає більше одного елемента, результуючий ітератор вказує на перший з таких елементів.

Див. приклад до `max_element`.

## **max\_element**

```
//default (1)
template <class ForwardIterator>
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

// custom (2)
template <class ForwardIterator, class Compare>
    ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                               Compare comp);
```

Найбільший елемент в діапазоні.

Повертає ітератор, що вказує на елемент з найбільшим значенням в діапазоні [first, last).

Порівняння виконуються за допомогою або `operator<` для першої версії, або `comp` для другої. Елемент є найбільшим, якщо немає інших елементів, не менших за нього. Якщо цій умові відповідає більше одного елемента, результуючий ітератор вказує на перший з таких елементів.

Приклад:

```
// min_element / max_element example
#include <iostream>      // std::cout
#include <algorithm>     // std::min_element, std::max_element

bool myfn(int i, int j) { return i < j; }

struct myclass {
    bool operator() (int i, int j) { return i < j; }
} myobj;

int main()
{
    int myints[] = { 3,7,2,5,6,4,9 };

    // using default comparison:
    std::cout << "The smallest element is "
               << *std::min_element(myints, myints + 7) << '\n';
```

```

std::cout << "The largest element is "
    << *std::max_element(myints, myints + 7) << '\n';

// using function myfn as comp:
std::cout << "The smallest element is "
    << *std::min_element(myints, myints + 7, myfn) << '\n';
std::cout << "The largest element is "
    << *std::max_element(myints, myints + 7, myfn) << '\n';

// using object myobj as comp:
std::cout << "The smallest element is "
    << *std::min_element(myints, myints + 7, myobj) << '\n';
std::cout << "The largest element is "
    << *std::max_element(myints, myints + 7, myobj) << '\n';
// The smallest element is 2
// The largest element is 9
// The smallest element is 2
// The largest element is 9
// The smallest element is 2
// The largest element is 9

return 0;
}

```

## minmax\_element

```

// default (1)
template <class ForwardIterator>
    std::pair<ForwardIterator, ForwardIterator>
        minmax_element(ForwardIterator first, ForwardIterator last);

// custom (2)
template <class ForwardIterator, class Compare>
    std::pair<ForwardIterator, ForwardIterator>
        minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);

```

Пара з найменшого і найбільшого елементів діапазону.

Повертає пару (об'єкт `std::pair`) з двох ітераторів, перший вказує на елемент з найменшим значенням в діапазоні `[first, last)`, а другий – на найбільший. Порівняння виконуються за допомогою або `operator<` для першої версії, або `comp` для другої.

Якщо більш ніж один з еквівалентних елементів має найменше значення, перший ітератор пари вказує на перший з таких елементів.

Якщо більш ніж один з еквівалентних елементів має найбільше значення, другий ітератор пари вказує на останній з таких елементів.

Приклад:

```

// minmax_element
#include <iostream>      // std::cout
#include <algorithm>     // std::minmax_element
#include <array>         // std::array

int main()
{
    std::array<int, 7> foo{ 3,7,2,9,5,8,6 };

    auto result = std::minmax_element(foo.begin(), foo.end());
}

```

```
// print result:
std::cout << "min is " << *result.first;
std::cout << ", at position " << (result.first - foo.begin()) << '\n';
std::cout << "max is " << *result.second;
std::cout << ", at position " << (result.second - foo.begin()) << '\n';
// min is 2, at position 2
// max is 9, at position 3

return 0;
}
```

## Лексикографічні операції

lexicographical_compare	Лексикографічне порівняння, чи менше
next_permutation	Перетворити діапазон на наступну перестановку
prev_permutation	Перетворити діапазон на попередню перестановку

Перестановка – це кожна з  $N!$  можливих схем розташування елементів (де  $N$  – кількість елементів у діапазоні). Можна впорядкувати різні перестановки залежно від того, як вони лексикографічно порівнюються між собою. Перша можлива таким чином відсортована перестановка (та, яка лексикографічно найменша порівняно з усіма іншими перестановками) – це та, яка має всі елементи відсортовані за зростанням, а найбільша – усі елементи, відсортовані за спаданням.

### lexicographical\_compare

```
// default (1)
template <class InputIterator1, class InputIterator2>
    bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2);

// custom (2)
template <class InputIterator1, class InputIterator2, class Compare>
    bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                                Compare comp);
```

Лексикографічне порівняння, чи менше.

Повертає `true`, якщо діапазон `[first1, last1)` лексикографічно менший, ніж діапазон `[first2, last2)`.

Лексикографічне порівняння – це порівняння, яке зазвичай використовується для сортування слів за алфавітом у словниках. Воно включає послідовне порівняння елементів, які мають однакове положення в обох діапазонах один з іншим, поки один елемент не стане не еквівалентний іншому. Результат порівняння цих перших невідповідних елементів є результатом лексикографічного порівняння. Якщо обидві послідовності мають однакові елементи у відповідних позиціях, то коротша послідовність лексикографічно менша за довшу.

Елементи порівнюються за допомогою `operator<` для першої версії та `comp` для другої. Два елементи, `a` і `b`, вважаються еквівалентними, якщо `!(a<b) && !(b<a)` або якщо `!(comp(a, b) && !comp(b, a))`.

Приклад:

```
// lexicographical_compare example
#include <iostream>      // std::cout, std::boolalpha
#include <algorithm>     // std::lexicographical_compare
#include <cctype>        // std::tolower
```



```

// a case-insensitive comparison function:
bool mycomp(char c1, char c2)
{
    return std::tolower(c1) < std::tolower(c2);
}

int main()
{
    char foo[] = "Apple";
    char bar[] = "apartment";

    std::cout << std::boolalpha;

    std::cout << "Comparing foo and bar lexicographically (foo<bar):\n";

    std::cout << "Using default comparison (operator<): ";
    std::cout << std::lexicographical_compare(foo, foo + 5, bar, bar + 9);
    std::cout << '\n';

    std::cout << "Using mycomp as comparison object: ";
    std::cout << std::lexicographical_compare(foo, foo + 5, bar, bar + 9, mycomp);
    std::cout << '\n';
    // Comparing foo and bar lexicographically (foo<bar):
    // Using default comparison (operator<): true
    // Using mycomp as comparison object: false

    return 0;
}

```

Порівняння за умовчанням порівнює звичайні ASCII-коди символів, де символ 'A' (код 65) менше, ніж символ 'a' (код 97).

Функція `mycomp()` перетворює літери до нижнього регістру перед їх порівнянням, тому тут перша буква, що не відповідає, є третьою ('a' проти 'p').

## next\_permutation

```

// default (1)
template <class BidirectionalIterator>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);

// custom (2)
template <class BidirectionalIterator, class Compare>
    bool next_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);

```

Перетворити діапазон у наступну перестановку.

Переставляє елементи в діапазоні `[first, last)` у наступну лексикографічно більшу перестановку.

Перестановка – це кожна з  $N!$  можливих схем розташування елементів (де  $N$  – кількість елементів у діапазоні). Можна впорядкувати різні перестановки залежно від того, як вони лексикографічно порівнюються між собою. Перша можлива таким чином відсортована перестановка (та, яка лексикографічно найменша порівняно з усіма іншими

перестановками) – це та, яка має всі елементи відсортовані за зростанням, а найбільша – усі елементи, відсортовані за спаданням.

Порівняння окремих елементів виконується за допомогою або `operator<` для першої версії, або `comp` для другої.

Якщо функція може визначити наступну більшу перестановку, вона переставляє елементи відповідним чином і повертає `true`. Якщо це – неможливо (оскільки поточна перестановка вже є лексикографічно найбільшою перестановкою), функція переставляє елементи відповідно до першої перестановки (тобто, лексикографічно найменшої, коли елементи відсортовані за зростанням) і повертає `false`.

Приклад:

```
// next_permutation example
#include <iostream>      // std::cout
#include <algorithm>     // std::next_permutation, std::sort

int main()
{
    int myints[] = { 1,2,3 };

    std::sort(myints, myints + 3);

    std::cout << "The 3! possible permutations with 3 elements:\n";

    do {
        std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
    } while (std::next_permutation(myints, myints + 3));

    std::cout << "After loop: " << myints[0] << ' '
              << myints[1] << ' ' << myints[2] << '\n';
    // The 3! possible permutations with 3 elements:
    // 1 2 3
    // 1 3 2
    // 2 1 3
    // 2 3 1
    // 3 1 2
    // 3 2 1
    // After loop : 1 2 3

    return 0;
}
```

## prev\_permutation

```
//default (1)
template <class BidirectionalIterator>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last);

// custom (2)
template <class BidirectionalIterator, class Compare>
    bool prev_permutation(BidirectionalIterator first,
                          BidirectionalIterator last, Compare comp);
```

Перетворити діапазон у попередню перестановку.

Переставляє елементи в діапазоні `[first, last)` у попередню лексикографічно меншу перестановку.

Перестановка – це кожна з  $N!$  можливих схем розташування елементів (де  $N$  – кількість елементів у діапазоні). Можна впорядкувати різні перестановки залежно від того, як вони лексикографічно порівнюються між собою. Перша можлива таким чином відсортована перестановка (та, яка лексикографічно найменша порівняно з усіма іншими перестановками) – це та, яка має всі елементи відсортовані за зростанням, а найбільша – усі елементи, відсортовані за спаданням.

Порівняння окремих елементів виконується за допомогою або `operator<` для першої версії, або `comp` для другої.

Якщо функція може визначити попередню перестановку, вона переставляє елементи відповідним чином і повертає `true`. Якщо це – неможливо (оскільки поточна перестановка вже є лексикографічно найменшою з можливих перестановок), функція переставляє елементи відповідно до останньої перестановки (тобто, лексикографічно найбільшої, коли елементи відсортовані за спаданням) і повертає значення `false`.

Приклад:

```
// prev_permutation example
#include <iostream>      // std::cout
#include <algorithm>     // std::next_permutation, std::sort, std::reverse

int main()
{
    int myints[] = { 1,2,3 };

    std::sort(myints, myints + 3);
    std::reverse(myints, myints + 3);

    std::cout << "The 3! possible permutations with 3 elements:\n";
    do {
        std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
    } while (std::prev_permutation(myints, myints + 3));

    std::cout << "After loop: " << myints[0] << ' '
              << myints[1] << ' ' << myints[2] << '\n';
    // The 3! possible permutations with 3 elements:
    // 3 2 1
    // 3 1 2
    // 2 3 1
    // 2 1 3
    // 1 3 2
    // 1 2 3
    // After loop : 3 2 1

    return 0;
}
```

# Лабораторний практикум

## Оформлення звіту про виконання лабораторних робіт

### Вимоги до оформлення звіту про виконання лабораторних робіт №№ 7.1–7.4

Звіт про виконання лабораторних робіт №№ 7.1–7.4 має містити наступні елементи:

- 1) заголовок;
- 2) мету роботи;
- 3) умову завдання;

*Умова завдання має бути вставлена у звіт як фрагмент зображення (скрін) сторінки посібника.*

- 4) UML-діаграму класів;
- 5) структурну схему програми;

*Структурна схема програми зображує взаємозв'язки програми та всіх її програмних одиниць: схему вкладеності та охоплення підпрограм, програми та модулів; а також схему звертання одних програмних одиниць до інших.*

- 6) текст програми;

*Текст програми має бути правильно відформатований: відступами і порожніми рядками слід відображати логічну структуру програми; програма має містити необхідні коментарі – про призначення підпрограм, змінних та параметрів – якщо їх імена не значущі, та про призначення окремих змістовних фрагментів програми. Текст програми слід подавати моноширинним шрифтом (Courier New розміром 10 пт. або Consolas розміром 9,5 пт.) з одинарним міжрядковим інтервалом;*

- 7) посилання на git-репозиторій з проектом (див. інструкції з Лабораторної роботи № 2.2 з предмету «Алгоритмізація та програмування»);
- 8) хоча б для одної функції, яка повертає результат (як результат функції чи як параметр-посилання) – результати unit-тесту: текст програми unit-тесту та скрін результатів її виконання (див. інструкції з Лабораторної роботи № 5.6 з предмету «Алгоритмізація та програмування»);
- 9) висновки.

## **Зразок оформлення звіту про виконання лабораторних робіт №№ 7.1–7.4**

### **ЗВІТ**

про виконання лабораторної роботи № < номер >

« назва теми лабораторної роботи »

з дисципліни

«Об'єктно-орієнтоване програмування»

студента(ки) групи КН-26

< Прізвище Ім'я По\_батькові >

#### **Мета роботи:**

...

#### **Умова завдання:**

...

#### **UML-діаграма класів:**

...

#### **Структурна схема програми:**

...

#### **Текст програми:**

...

#### **Посилання на git-репозиторій з проектом:**

...

#### **Результати unit-тесту:**

...

#### **Висновки:**

...

# Лабораторна робота № 7.1. Контейнери-масиви

## Мета роботи

Навчитися використовувати стандартні контейнери-масиви.

## Питання, які необхідно вивчити та пояснити на захисті

- 1) Типи стандартних колекцій та їх призначення
- 2) Загальний синтаксис оголошення колекцій-масивів.
- 3) Наповнення та доступ до елементів колекції-масиву.
- 4) Дії з елементами колекції-масиву.
- 5) Загальні схеми циклів опрацювання колекцій-масивів для пошуку, обчислення кількості та суми заданих елементів.

## Приклад виконання завдання

### Спосіб 1.

```
// обчислити середнє арифметичне непарних елементів
// спосіб 1 - обчислення "вручну"
#include <iostream>
#include <array>
#include <ctime>

using namespace std;

void create(array<int, 100>& arr, int n)
{
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100;
}

void print(array<int, 100> arr, int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

double avg(array<int, 100> arr, int n)
{
    int count = 0;
    double summ = 0;
    for (int i = 0; i < n; i++)
        if (arr[i] % 2 != 0)
        {
            summ += arr[i];
            count++;
        }
    return summ / count;
}
```

```

int main()
{
    srand((unsigned int)time(NULL));
    int n;
    cout << "n = ? "; cin >> n;
    array<int, 100> arr = { 0 };

    create(arr, n);
    print(arr, n);
    cout << "AVG = " << avg(arr, n) << endl;

    return 0;
}

```

## Спосіб 2.

```

// обчислити середнє арифметичне непарних елементів
// спосіб 2 - використання алгоритмів STL
#include <iostream>      // std::cout
#include <algorithm>     // std::generate
#include <vector>         // std::vector
#include <ctime>          // std::time
#include <numeric>        // std::accumulate

using namespace std;

// function generator:
int RandomNumber() { return (std::rand() % 100); }

void print(vector<int> a)
{
    for (int i : a)
        cout << i << " ";
    cout << endl;
}

int main()
{
    srand(unsigned(time(0)));

    int n;
    cout << "n = ? "; cin >> n;
    vector<int> v(n);

    generate(v.begin(), v.end(), RandomNumber);
    print(v);

    vector<int> u(v.size());
    auto p = copy_if(v.begin(), v.end(), u.begin(), [](int i) {return i % 2 != 0; });

    u.resize(p - u.begin());
    print(u);

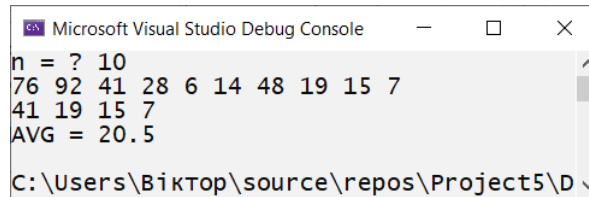
    auto k = u.size();
    auto s = accumulate(u.begin(), u.end(), 0);

    cout << "AVG = " << s * 1. / k << endl;

    return 0;
}

```

Результат виконання:



```
Microsoft Visual Studio Debug Console
n = ? 10
76 92 41 28 6 14 48 19 15 7
41 19 15 7
AVG = 20.5
C:\Users\Віктор\source\repos\Project5\D
```

## Варіанти завдань

Необхідно написати програму для того, щоб виконати наступні дії:

- сформувати колекцію-масив;
- вивести її на екран у вигляді рядка;
- виконати вказані у завдання дії;
- вивести результат, причому, якщо колекція-масив була змінена – то вивести на екран модифіковану колекцію-масив у вигляді наступного рядка.

Всі вказані дії необхідно реалізувати за допомогою окремих підпрограм. Інформація у підпрограми повинна передаватися лише за допомогою параметрів. Використання глобальних змінних – не допускається.

### Варіант 1.

Написати підпрограму, яка міняє місцями максимальний та мінімальний елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### Варіант 2.

Написати підпрограму, яка шукає індекс найменшого непарного елемента одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### Варіант 3.

Написати підпрограму, яка обчислює кількість непарних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### Варіант 4.

Написати підпрограму, яка обчислює середнє арифметичне індексів максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### Варіант 5.

Написати підпрограму, яка обчислює середнє арифметичне непарних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.



### **Варіант 6.**

Написати підпрограму, яка міняє місцями перший елемент із найменшим парним елементом одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 7.**

Написати підпрограму, яка шукає максимальний та мінімальний елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 8.**

Написати підпрограму, яка міняє місцями елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу так, щоб вони розмістилися в зворотному порядку:  $a_n, a_{n-1}, \dots, a_2, a_1$ .

### **Варіант 9.**

Написати підпрограму, яка обчислює суму індексів непарних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 10.**

Написати підпрограму, яка обчислює середнє арифметичне елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу з парними індексами.

### **Варіант 11.**

Написати підпрограму, яка міняє місцями останній елемент із найбільшим непарним елементом одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 12.**

Написати підпрограму, яка шукає індекси найбільшого та найменшого елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 13.**

Написати підпрограму, яка обчислює середнє арифметичне максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 14.**

Написати підпрограму, яка обчислює суму елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу з непарними індексами.

### **Варіант 15.**

Написати підпрограму, яка обчислює суму максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 16.**

Написати підпрограму, яка шукає найменший парний елемент одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 17.**

Написати підпрограму, яка шукає індекс найбільшого парного елемента одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 18.**

Написати підпрограму, яка обчислює суму індексів максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 19.**

Написати підпрограму, яка шукає найбільший непарний елемент одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 20.**

Написати підпрограму, яка обчислює середнє арифметичне індексів парних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 21.**

Написати підпрограму, яка міняє місцями елементи одновимірного масиву (вектора)  $a$  із  $2n$  елементів цілого типу так, щоб вони розмістилися в такому порядку:  $a_2, a_1, a_4, a_3, \dots, a_{2n}, a_{2n-1}$ . (кожний елемент з парним індексом міняється місцями з попереднім елементом).

### **Варіант 22.**

Написати підпрограму, яка обчислює суму парних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 23.**

Написати підпрограму, яка міняє місцями елементи одновимірного масиву (вектора)  $a$  із  $2n$  елементів цілого типу так, щоб вони розмістилися в такому порядку:  $a_{n+1}, \dots, a_{2n}, a_1, \dots, a_n$  (перша половина елементів вектора міняється місцями з другою).

### **Варіант 24.**

Написати підпрограму, яка впорядковує по спаданню елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 25.**

Написати підпрограму, яка обчислює середнє арифметичне максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 26.**

Написати підпрограму, яка міняє місцями максимальний та мінімальний елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 27.**

Написати підпрограму, яка шукає індекс найменшого непарного елемента одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 28.**

Написати підпрограму, яка обчислює кількість непарних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 29.**

Написати підпрограму, яка обчислює середнє арифметичне індексів максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 30.**

Написати підпрограму, яка обчислює середнє арифметичне непарних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 31.**

Написати підпрограму, яка міняє місцями перший елемент із найменшим парним елементом одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 32.**

Написати підпрограму, яка шукає максимальний та мінімальний елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 33.**

Написати підпрограму, яка міняє місцями елементи одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу так, щоб вони розмістилися в зворотному порядку:  $a_n, a_{n-1}, \dots, a_2, a_1$ .

### **Варіант 34.**

Написати підпрограму, яка обчислює суму індексів непарних елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 35.**

Написати підпрограму, яка обчислює середнє арифметичне елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу з парними індексами.

### **Варіант 36.**

Написати підпрограму, яка міняє місцями останній елемент із найбільшим непарним елементом одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 37.**

Написати підпрограму, яка шукає індекси найбільшого та найменшого елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 38.**

Написати підпрограму, яка обчислює середнє арифметичне максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.

### **Варіант 39.**

Написати підпрограму, яка обчислює суму елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу з непарними індексами.

### **Варіант 40.**

Написати підпрограму, яка обчислює суму максимального та мінімального елементів одновимірного масиву (вектора)  $a$  із  $n$  елементів цілого типу.



## Лабораторна робота № 7.2. Контейнери-списки

### Мета роботи

Навчитися використовувати стандартні контейнери-списки.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Типи стандартних колекцій та їх призначення
- 2) Загальний синтаксис оголошення колекцій-списків.
- 3) Наповнення та доступ до елементів колекції-списку.
- 4) Дії з елементами колекції-списку.
- 5) Загальні схеми циклів опрацювання колекцій-списків для пошуку, обчислення кількості та суми заданих елементів.

### Приклад виконання завдання

#### Спосіб 1.

```
// обчислити кількість від'ємних елементів
// спосіб 1 - обчислення "вручну"
#include <iostream>
#include <list>
#include <ctime>

using namespace std;

void create(list<int>& list, int n)
{
    for (int i = 0; i < n; i++)
        list.push_back(rand() % 100 - 50);
}

void print(list<int> list, int n)
{
    for (int i = 0; i < n; i++)
    {
        int k = list.front();
        list.pop_front();
        cout << k << " ";
        list.push_back(k);
    }
    cout << endl;
}

double count(list<int> list, int n)
{
    int count = 0;
    for (int i = 0; i < n; i++)
    {
        int k = list.front();
        list.pop_front();
    }
}
```

```

        if (k < 0)
        {
            count++;
        }
        list.push_back(k);
    }
    return count;
}

int main()
{
    srand((unsigned int)time(NULL));
    int n;
    cout << "n = ? "; cin >> n;
    list<int> arr;

    create(arr, n);
    print(arr, n);
    cout << "negative = " << count(arr, n) << endl;

    return 0;
}

```

## Спосіб 2.

```

// обчислити кількість від'ємних елементів
// спосіб 2 - використання алгоритмів STL
#include <iostream> // std::cout
#include <list> // std::list
#include <algorithm> // std::generate
#include <ctime> // std::time
#include <numeric> // std::accumulate

using namespace std;

// function generator:
int RandomNumber() { return rand() % 100 - 50; }

void print(list<int> a)
{
    for (int i : a)
        cout << i << " ";
    cout << endl;
}

int main()
{
    srand((unsigned int)time(NULL));
    int n;
    cout << "n = ? "; cin >> n;
    list<int> l(n);

    generate(l.begin(), l.end(), RandomNumber);
    print(l);

    auto k = count_if(l.begin(), l.end(), [](int i) {return i < 0; });
    cout << "count of negative = " << k << endl;

    return 0;
}

```

## **Варіанти завдань**

Виконати свій варіант за допомогою відповідних стандартних колекцій.

Всі завдання слід реалізувати з використанням лінійних однонаправлених списків. Якщо при виконанні завдання слід утворити нові списки, – вони повинні бути тої самої структури.

Якщо у завданні не вказано тип даних інформаційного поля, його слід вибрати самостійно.

Необхідно:

- сформулювати задану колекцію-список (списки);
- роздрукувати його (їх) – вивести значення елементів на екран;
- виконати вказані в завданні дії;
- вивести результат на екран.

Всі вказані дії слід виконувати в окремих підпрограмах, які всю інформацію приймають у вигляді параметрів. Використовувати нелокальні змінні не можна.

### **Варіант 1.**

Обчислити кількість елементів списку з непарними значеннями інформаційного поля.

### **Варіант 2.**

Обчислити кількість елементів списку із заданим користувачем значенням інформаційного поля.

### **Варіант 3.**

Обчислити суму елементів списку із парними значеннями інформаційного поля.

### **Варіант 4.**

Обчислити суму елементів списку із додатними значеннями інформаційного поля.

### **Варіант 5.**

Обчислити кількість елементів списку із від'ємними значеннями інформаційного поля.

### **Варіант 6.**

Збільшити значення інформаційного поля кожного елемента списку на задану користувачем величину.



### **Варіант 7.**

Продублювати кожний елемент списку із заданим користувачем значенням інформаційного поля.

### **Варіант 8.**

Вилучити кожний елемент списку із заданим користувачем значенням інформаційного поля.

### **Варіант 9.**

Вилучити кожний елемент списку, який передує елементу із заданим користувачем значенням інформаційного поля.

### **Варіант 10.**

Вилучити кожний елемент списку, який слідує за елементом із заданим користувачем значенням інформаційного поля.

### **Варіант 11.**

Перед кожним елементом списку із значенням інформаційного поля V1 вставити новий елемент із значенням інформаційного поля V2.

### **Варіант 12.**

Після кожного елементу списку із значенням інформаційного поля V1 вставити новий елемент із значенням інформаційного поля V2.

### **Варіант 13.**

Перевірити, чи елементи списку – впорядковані за не спаданням.

### **Варіант 14.**

Визначити, чи список містить пару сусідніх елементів з однаковими значеннями інформаційного поля.

### **Варіант 15.\***

Визначити, чи список містить пару елементів (не обов'язково сусідніх) з однаковими значеннями інформаційного поля.

### **Варіант 16.\***

Визначити, чи всі елементи списку  $L_1$  входять у список  $L_2$  (тобто, чи список  $L_2$  містить всі елементи списку  $L_1$ , розташовані у довільному порядку).

### **Варіант 17.\***

Визначити, чи список  $L_1$  входить як частина у список  $L_2$  (тобто, чи список  $L_2$  містить всі елементи списку  $L_1$ , розташовані у тому ж самому порядку).

### **Варіант 18.**

На основі списку  $L$  створити два списки:  $L_1$  – із елементів з додатними значеннями інформаційного поля та  $L_2$  – із елементів з від’ємними значеннями інформаційного поля, зберігши порядок їх розташування.

### **Варіант 19.**

Поміняти місцями елементи списку – змінити порядок розташування на зворотний.

### **Варіант 20.**

Поміняти місцями елементи списку – перший з другим, третій з четвертим і т.д.

### **Варіант 21.\*\***

Вилучити із списку всі елементи-дублікати (елементи, що повторюються).

### **Варіант 22.\*\***

Вилучити із списку всі унікальні елементи (всі елементи, що входять лише один раз), залишити лише елементи-дублікати.

### **Варіант 23.\*\***

Обчислити кількість елементів-дублікатів списку (елементів, що повторюються).

### **Варіант 24.\*\***

Обчислити кількість унікальних елементів списку (елементів, що входять лише один раз).

### **Варіант 25.\***

Дано два списки  $L_1$  та  $L_2$  – елементи яких впорядковані.

Об’єднати ці списки, створивши список  $L$ , зберігши впорядкованість.

Наприклад:  $L_1 = \{1, 4, 5\}$ ;  $L_2 = \{2, 3, 6\}$ . Тоді  $L = \{1, 2, 3, 4, 5, 6\}$ .

### **Варіант 26.**

Обчислити кількість елементів списку з непарними значеннями інформаційного поля.

### **Варіант 27.**

Обчислити кількість елементів списку із заданим користувачем значенням інформаційного поля.

### **Варіант 28.**

Обчислити суму елементів списку із парними значеннями інформаційного поля.

### **Варіант 29.**

Обчислити суму елементів списку із додатними значеннями інформаційного поля.

### **Варіант 30.**

Обчислити кількість елементів списку із від'ємними значеннями інформаційного поля.

### **Варіант 31.**

Збільшити значення інформаційного поля кожного елемента списку на задану користувачем величину.

### **Варіант 32.**

Продублювати кожний елемент списку із заданим користувачем значенням інформаційного поля.

### **Варіант 33.**

Вилучити кожний елемент списку із заданим користувачем значенням інформаційного поля.

### **Варіант 34.**

Вилучити кожний елемент списку, який передує елементу із заданим користувачем значенням інформаційного поля.

### **Варіант 35.**

Вилучити кожний елемент списку, який слідує за елементом із заданим користувачем значенням інформаційного поля.

**Варіант 36.**

Перед кожним елементом списку із значенням інформаційного поля V1 вставити новий елемент із значенням інформаційного поля V2.

**Варіант 37.**

Після кожного елемента списку із значенням інформаційного поля V1 вставити новий елемент із значенням інформаційного поля V2.

**Варіант 38.**

Перевірити, чи елементи списку – впорядковані за не спаданням.

**Варіант 39.**

Визначити, чи список містить пару сусідніх елементів з однаковими значеннями інформаційного поля.

**Варіант 40.\***

Визначити, чи список містить пару елементів (не обов'язково сусідніх) з однаковими значеннями інформаційного поля.

## Лабораторна робота № 7.3. Контейнери-множини

### Мета роботи

Навчитися використовувати стандартні контейнери-множини.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Типи стандартних колекцій та їх призначення
- 2) Загальний синтаксис оголошення колекцій-множин.
- 3) Наповнення та доступ до елементів колекції-множини.
- 4) Дії з елементами колекції-множини.
- 5) Загальні схеми циклів опрацювання колекції-множини для пошуку, обчислення кількості та суми заданих елементів.

### Приклад виконання завдання

#### Спосіб 1.

```
// побудувати множину символів A-F,  
// які є хоча б в одному із двох введених літерних рядків  
// спосіб 1 - обчислення "вручну"  
#include <iostream>  
#include <string>  
#include <set>  
#include <Windows.h> // підключаємо бібліотеку, яка забезпечує відображення кирилиці  
  
using namespace std;  
  
string toStr(set<string> s)  
{  
    set<string>::iterator i;  
    string str = "{";  
    for (i = s.begin(); i != s.end(); i++)  
        str += *i + " ";  
  
    str = str.erase(str.length() - 1);  
    str += "}";  
    return str;  
}  
  
set<string> create(string t)  
{  
    set<string> s;  
    string c;  
    for (int i = 0; i < t.length(); i++)  
    {  
        c = t[i];  
        s.insert(c);  
    }  
    return s;  
}
```

```

// Увага! Для обчислення різниці множин слід використовувати стандартний алгоритм STL
// (ця програма створювалася ще тоді, коли в STL не було відповідного алгоритму)
// різниця множин
set<string> difference(set<string> s1, set<string> s2)
{
    // копія першої множини
    set<string> s = s1;
    set<string>::iterator i, i2;

    // з s1 вилучаємо символи s2
    for (i2 = s2.begin(); i2 != s2.end(); i2++)
    {
        i = s.find(*i2);
        if (i != s.end())
            s.erase(*i);
    }
    // отримали: символи s1, які не входять в s2
    return s;
}

// використання операції різниці множин
set<string> calc(set<string> s1, set<string> s2, set<string> s0)
{
    set<string> s;
    s = difference(s0, s1); // з базової вилучаємо символи s1
    s = difference(s, s2);  // з базової вилучаємо символи s2
    // отримали: символи базової, які не входять в s1 та s2

    set<string> res;
    res = difference(s0, s); // з базової вилучаємо символи s
    // отримали: символи базової, які входять в s1 або в s2
    return res;
}

void print(set<string> s)
{
    set<string>::iterator i;
    for (i = s.begin(); i != s.end(); i++)
    {
        cout << *i << " ";
    }
    cout << endl;
}

double count(set<string> s)
{
    int count = 0;
    set<string>::iterator i;
    for (i = s.begin(); i != s.end(); i++)
    {
        if (*i == "a")
        {
            count++;
        }
    }
    return count;
}

int main()
{
    SetConsoleCP(1251); // встановлення кодової сторінки win-cp1251

```

```

// (кирилиця) в потік вводу
SetConsoleOutputCP(1251); // встановлення кодової сторінки win-cp1251
// (кирилиця) в потік виводу

// вводим 1-й літерний рядок
string t1;
cout << "перша послідовність символів: ";
getline(cin, t1);

// вводим 2-й літерний рядок
string t2;
cout << "друга послідовність символів: ";
getline(cin, t2);

// перша множина
set<string> s1 = create(t1);
cout << "перша множина: " + toString(s1) << endl;

// друга множина
set<string> s2 = create(t2);
cout << "друга множина: " + toString(s2) << endl;

// базова множина
set<string> s0 = create("ABCDEF");
cout << "базова множина: " + toString(s0) << endl;

// результуюча множина
set<string> s;
s = calc(s1, s2, s0);

// виводимо результуючу множину на екран
cout << "результуюча множина: " + toString(s) << endl;

return 0;
}

```

## Спосіб 2.

```

// побудувати множину символів A-F,
// які є хоча б в одному із двох введених літерних рядків
// спосіб 2 - використання алгоритмів STL
#include <iostream>
#include <sstream>
#include <string>
#include <set>
#include <algorithm>
#include <Windows.h> // підключаємо бібліотеку, яка забезпечує відображення кирилиці

using namespace std;

void print(set<char> s)
{
    stringstream ss;
    ss << "{";
    for (auto i : s)
        ss << i << ";";

    string st = ss.str();
    st = st.erase(st.length() - 1) + "}";
    cout << st << endl;
}

```

```

set<char> difference(set<char> a, set<char> b)
{
    set<char> r(a); // r = a
    for (auto v : b)
        r.erase(v); // r = a - b
    return r;
}

// використання операцій над множинами
set<char> calc(set<char> s1, set<char> s2, set<char> s0)
{
    set<char> s = difference(s0, s1); // s = s0 - s1
    s = difference(s, s2);           // s = s - s2
    // отримали: символи базової, які не входять в s1 та s2
    // s = s0 - s1 - s2

    set<char> r = difference(s0, s); // r = s0 - s
    // отримали: символи базової, які входять в s1 або в s2
    // r = s0 - (s0 - s1 - s2)
    // r = s0 * (s1 + s2)

    return r;
}

int count(set<char> s)
{
    return std::count(s.begin(), s.end(), 'a');
}

int main()
{
    // забезпечення відображення кирилиці:
    SetConsoleCP(1251);           // встановлення кодової сторінки win-cp1251
                                   // (кирилиця) в потік вводу
    SetConsoleOutputCP(1251);     // встановлення кодової сторінки win-cp1251
                                   // (кирилиця) в потік виводу

    string t1;                    // вводимо 1-й літерний рядок
    cout << "перша послідовність символів: "; getline(cin, t1);

    set<char> s1(t1.begin(), t1.end()); // перша множина
    cout << "перша множина: " << endl; print(s1);

    string t2;                    // вводимо 2-й літерний рядок
    cout << "друга послідовність символів: "; getline(cin, t2);

    set<char> s2(t2.begin(), t2.end()); // друга множина
    cout << "друга множина: " << endl; print(s2);

    string t0 = "ABCDEF";         // базова множина
    set<char> s0(t0.begin(), t0.end());
    cout << "базова множина: " << endl; print(s0);

    set<char> s;                  // результуюча множина
    s = calc(s1, s2, s0);
    cout << "результуюча множина: " << endl; print(s);

    return 0;
}

```



## **Варіанти завдань**

Завдання необхідно виконувати за допомогою операцій над множинами.

Для кожного завдання слід нарисувати діаграму Ейлера-Венна, яка пояснює побудову шуканої множини.

Опрацювання множин (кожну вказану у завданні дію) слід виконувати у підпрограмах, які всю інформацію приймають у вигляді параметрів. Використовувати нелокальні змінні не можна.

### **Варіант 1.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘O’ до ‘S’,

що не зустрічаються в жодній із двох заданих послідовностей символів.

### **Варіант 2.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘O’ до ‘S’,

що одночасно зустрічаються в двох заданих послідовностях символів.

### **Варіант 3.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 4.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки арифметичних операцій ‘+’, ‘-’, ‘\*’, ‘/’,

що зустрічаються в заданому тексті лише один раз.

### **Варіант 5.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘A’ до ‘F’,

що зустрічаються хоча би в одній із двох заданих послідовностей символів.

### **Варіант 6.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

цифри від ‘0’ до ‘9’,

що не зустрічаються в жодній із двох заданих послідовностей символів.

### **Варіант 7.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘G’ до ‘N’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 8.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки арифметичних операцій: ‘<’, ‘>’, ‘=’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 9.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘A’ до ‘F’,

що одночасно зустрічаються в двох заданих послідовностях символів.

### **Варіант 10.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки пунктуації ‘,’ , ‘.’, ‘:’, ‘;’, ‘!’, ‘?’ ,

що зустрічаються в заданому тексті лише один раз.

### **Варіант 11.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘A’ до ‘F’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 12.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

цифри від ‘0’ до ‘9’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 13.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 14.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що не зустрічаються в заданому тексті.

### **Варіант 15.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

дужки ‘(’, ‘)’, ‘[’, ‘]’, ‘{’, ‘}’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 16.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘T’ до ‘Z’,

що не зустрічаються в заданому тексті.

### **Варіант 17.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки пунктуації ‘,’, ‘.’, ‘:’, ‘;’, ‘!’, ‘?’,

що одночасно зустрічаються в двох заданих послідовностях символів.

### **Варіант 18.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘G’ до ‘N’,

що зустрічаються в заданому тексті лише один раз.

### **Варіант 19.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки операцій відношення: ‘<’, ‘>’, ‘=’,

що зустрічаються хоча би в одній із двох заданих послідовностей символів.

### **Варіант 20.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘O’ до ‘S’,

що не зустрічаються в заданому тексті.

### **Варіант 21.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки пунктуації ‘,’ , ‘.’, ‘:’, ‘;’, ‘!’, ‘?’ ,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 22.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘G’ до ‘N’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 23.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що зустрічаються хоча би в одній із двох заданих послідовностей символів.

#### **Варіант 24.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

дужки ‘(’, ‘)’ , ‘[’, ‘]’, ‘{’, ‘}’,

що не зустрічаються в жодній із двох заданих послідовностей символів.

#### **Варіант 25.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘O’ до ‘S’,

що не зустрічаються в жодній із двох заданих послідовностей символів.

#### **Варіант 26.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘O’ до ‘S’,

що одночасно зустрічаються в двох заданих послідовностях символів.

#### **Варіант 27.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

#### **Варіант 28.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки арифметичних операцій ‘+’, ‘-’, ‘\*’, ‘/’,

що зустрічаються в заданому тексті лише один раз.

#### **Варіант 29.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘A’ до ‘F’,

що зустрічаються хоча би в одній із двох заданих послідовностей символів.

### **Варіант 30.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

цифри від ‘0’ до ‘9’,

що не зустрічаються в жодній із двох заданих послідовностей символів.

### **Варіант 31.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘G’ до ‘N’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 32.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки арифметичних операцій: ‘<’, ‘>’, ‘=’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 33.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘A’ до ‘F’,

що одночасно зустрічаються в двох заданих послідовностях символів.

### **Варіант 34.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

знаки пунктуації ‘,’ , ‘.’, ‘:’, ‘;’, ‘!’, ‘?’ ,

що зустрічаються в заданому тексті лише один раз.

### **Варіант 35.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘A’ до ‘F’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 36.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

цифри від ‘0’ до ‘9’,

що зустрічаються в першій заданій послідовності символів і не зустрічаються – в другій.

### **Варіант 37.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 38.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

спеціальні символи ‘@’, ‘^’, ‘#’, ‘&’,

що не зустрічаються в заданому тексті.

### **Варіант 39.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

дужки ‘(’, ‘)’, ‘[’, ‘]’, ‘{’, ‘}’,

що зустрічаються в заданому тексті не менше, ніж два рази.

### **Варіант 40.**

Побудувати, визначити кількість елементів і роздрукувати множину символів, елементи якої – це:

букви від ‘T’ до ‘Z’,

що не зустрічаються в заданому тексті.

## Лабораторна робота № 7.4. Контейнери-відображення

### Мета роботи

Навчитися використовувати стандартні контейнери-відображення.

### Питання, які необхідно вивчити та пояснити на захисті

- 1) Типи стандартних колекцій та їх призначення
- 2) Загальний синтаксис оголошення колекцій-відображень.
- 3) Наповнення та доступ до елементів колекції-відображення.
- 4) Дії з елементами колекції-відображення.
- 5) Загальні схеми циклів опрацювання колекції-відображення для пошуку заданих елементів.

### Приклад виконання завдання

#### Варіант 0.

Розклад електричок зберігається у вигляді колекції. Кожен елемент містить назву пункту призначення, позначки типу «звичайний», «підвищеного комфорту», «швидкісний експрес» та час відправлення.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- виводить на екран інформації про поїзди, що відходять після введеного часу.

#### Розв'язок

```
#include <iostream>
#include <string>
#include <map>
#include <ctime>
#include <Windows.h> // підключаємо бібліотеку, яка забезпечує відображення кирилиці

using namespace std;

struct Train
{
    string city;
    int type;
    int hour;
    int minute;
};

void add(map<int, Train> &c)
{
    int pos = c.size();
```



```

int tmp2;
Train tmp;
cout << "Місто відправлення : ";
cin >> tmp.city;

cout << "[1] Звичайний" << endl;
cout << "[2] Підвищеного комфорту" << endl;
cout << "[3] Швидкісний експрес" << endl;
do
{
    cin >> tmp2;
} while (tmp2 < 1 || tmp2 > 3);
tmp.type = tmp2;

cout << "Година відправлення : ";
do
{
    cin >> tmp2;
} while (tmp2 < 0 || tmp2 > 23);
tmp.hour = tmp2;

cout << "Хвилина відправлення : ";
do
{
    cin >> tmp2;
} while (tmp2 < 0 || tmp2 > 59);
tmp.minute = tmp2;

c.insert({ pos, tmp });
}

void display(map<int, Train> c)
{
    for (int i = 0; i < c.size(); i++)
    {
        cout << "Місто відправлення : " << c[i].city << endl;
        if (c[i].type == 1)
            cout << "Тип      : Звичайний" << endl;
        else
            if (c[i].type == 2)
                cout << "Тип      : Підвищеного комфорту" << endl;
            else
                cout << "Тип      : Швидкісний експрес" << endl;

        cout << "Година відправлення : " << c[i].hour << endl;
        cout << "Хвилина відправлення : " << c[i].minute << endl;
    }
}

void display(map<int, Train> c, int j)
{
    for (int i = j; i < c.size(); i++)
    {
        cout << "Місто відправлення : " << c[i].city << endl;
        if (c[i].type == 1)
            cout << "Тип      : Звичайний" << endl;
        else
            if (c[i].type == 2)
                cout << "Тип      : Підвищеного комфорту" << endl;
            else
                cout << "Тип      : Швидкісний експрес" << endl;
    }
}

```

```

        cout << "Година відправлення : " << c[i].hour << endl;
        cout << "Хвилина відправлення : " << c[i].minute << endl;
    }
}

void display(map<int, Train> c, int h, int m)
{
    for (int i = 0; i < c.size(); i++)
    {
        if (c[i].hour > h || (c[i].hour == h && c[i].minute > m))
        {
            display(c, i);
            return;
        }
    }
}

void sort(map<int, Train> &c)
{
    for (int i = 0; i < c.size() - 1; i++)
        for (int j = i; j < c.size(); j++)
        {
            if (c[i].hour > c[j].hour ||
                (c[i].hour == c[j].hour && c[i].minute > c[j].minute))
            {
                Train tmp;
                tmp = c[i];
                c.erase(i);
                c.insert({ i, c[j] });
                c.erase(j);
                c.insert({ j, tmp });
            }
        }
}

int main()
{
    // забезпечення відображення кирилиці:
    SetConsoleCP(1251);           // встановлення кодової сторінки win-cp1251
                                   // (кирилиця) в потік вводу
    SetConsoleOutputCP(1251);     // встановлення кодової сторінки win-cp1251
                                   // (кирилиця) в потік виводу

    map<int, Train> rozklad;

    int tmp2, tmp3, tmp4;
    do
    {
        cout << "[1] Додати запис" << endl;
        cout << "[2] Відобразити записи" << endl;
        cout << "[3] Вивести відповідно до часу" << endl;
        cout << "[0] Вийти" << endl;
        do
        {
            cin >> tmp2;
        } while (tmp2 < 0 || tmp2 > 3);

        switch (tmp2)
        {
            case 1:
                add(rozklad);
                sort(rozklad);
                break;

```

```

    case 2:
        display(rozklad);
        break;
    case 3:
        cout << "Година відправлення : ";
        do
        {
            cin >> tmp3;
        } while (tmp3 < 0 || tmp3 > 23);

        cout << "Хвилина відправлення : ";
        do
        {
            cin >> tmp4;
        } while (tmp4 < 0 || tmp4 > 59);

        display(rozklad, tmp3, tmp4);
        break;
    default:
        break;
}

} while (tmp2 != 0);

return 0;
}

```

## **Варіанти завдань**

Кожна програма повинна містити меню. Необхідно передбачити контроль помилок користувача при введенні даних.

Необхідні програмі дані слід реалізувати за допомогою колекції-відображення. Всі дії над даними слід виконувати в колекції.

При розробці програми застосувати технологію низхідного проектування. Логічно закінчені фрагменти оформити у вигляді підпрограм, всі необхідні дані яким передаються через список параметрів. Використовувати глобальні змінні – не можна.

### **Варіант 1.**

В колекції записана звітна відомість результатів екзаменаційної сесії студентської групи, яка для кожного студента містить прізвище, ініціали і оцінки з п'яти предметів.

Скласти програму, за допомогою якої можна створювати, переглядати та поповнювати колекцію і отримувати:

- список всіх студентів;
- список студентів, що склали іспити тільки на «5»;
- список студентів, що мають трійки;
- список студентів, що мають двійки. При цьому студент, що має більш ніж одну двійку, виключається із списку.

## Варіант 2.

Підприємство має місцеву телефонну станцію. В колекції записано телефонний довідник даного підприємства, який для кожного номера телефону містить номер приміщення і список службовців, що сидять в даному приміщенні.

Скласти програму, яка:

- створює, переглядає та поповнює базу;
- за номером телефону видає номер приміщення і список людей, що сидять в ньому;
- за номером приміщення видає номер телефону;
- за прізвищем службовця видає номер телефону і номер приміщення.

Номер телефону – двозначний. У одному приміщенні може знаходитися від одного до чотирьох службовців.

## Варіант 3.

У готелі є 15 номерів, з них 5 одномісних і 10 двомісних. Скласти програму, яка створює, переглядає та поповнює колекцію, що містить дані про мешканців і за прізвищем визначає номер, де проживає мешканець.

Програма запрошує прізвище мешканця.

- Якщо мешканця з таким прізвищем немає, про це видається повідомлення.
- Якщо мешканець з таким прізвищем в готелі єдиний, програма видає прізвище мешканця і номер помешкання.
- Якщо в готелі проживає два або більше мешканців з таким прізвищем, програма додатково запрошує ініціали.

## Варіант 4.

Є колекція, що містить список службовців. Для кожного службовця вказано прізвище і ініціали, назву посади, рік прийому на роботу і оклад (величину заробітної плати).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про службовця, прізвище якого введено з клавіатури.

## Варіант 5.

Розклад електричок зберігається у вигляді колекції. Кожен елемент містить назву пункту відправлення, назву пункту призначення, позначки типу «звичайний», «підвищеного комфорту», «швидкісний експрес» та час відправлення.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- виводить на екран інформації про поїзди, що відходять після введеного часу.
- виводить на екран інформації про поїзди, що відходять із заданого пункту відправлення.
- виводить на екран інформації про поїзди, що відходять до заданого пункту призначення.

## Варіант 6.

Є колекція, що містить список товарів. Для кожного товару вказані його назва, вартість одиниці товару в гривнях, кількість і одиниця вимірювання (наприклад, кількість: 100 шт., одиниця вимірювання: упаковка по 20 кг.).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товар, назва якого введена з клавіатури;
- здійснює вивід на екран інформації про товари із заданого з клавіатури діапазону вартості.

## Варіант 7.

Є колекція, що містить список товарів. Для кожного товару вказані його назва, назва магазину, в якому продається товар, вартість одиниці товару в гривнях і його кількість з вказівкою одиниці вимірювання (наприклад, кількість: 100 шт., одиниця вимірювання: упаковка по 20 кг.).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товар, назва якого введена з клавіатури;
- здійснює вивід на екран інформації про товари, що продаються в магазині, назва якого введена з клавіатури.

### **Варіант 8.**

Є колекція, що містить список студентської групи. Кожен елемент містить прізвище студента і три екзаменаційні оцінки, причому список ніяк не впорядкований.

Скласти програму, яка створює, переглядає та поповнює колекцію.

### **Варіант 9.**

Є колекція, що містить список товарів. Для кожного товару вказані його назва, назва магазину, в якому продається товар, вартість одиниці товару в гривнях і його кількість з вказівкою одиниці вимірювання (наприклад, кількість: 100 шт., одиниця вимірювання: упаковка по 20 кг.).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товари, що продаються в магазині, назва якого введена з клавіатури;
- здійснює вивід на екран інформації про товари із заданого з клавіатури діапазону вартості.

### **Варіант 10.**

Є колекція, що містить список маршрутів, кожний елемент містить наступні дані:

- назву початкового пункту маршруту;
- назву кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про маршрут, номер якого введений з клавіатури; якщо такого маршруту немає, вивести на екран відповідне повідомлення.

### **Варіант 11.**

Є колекція, що містить список маршрутів, кожний елемент містить наступні дані:

- назву початкового пункту маршруту;
- назву кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;

- здійснює вивід на екран інформації про маршрути, які починаються або закінчуються в пункті, назва якого введена з клавіатури; якщо таких маршрутів немає, вивести на екран відповідне повідомлення.

## **Варіант 12.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людину, номер телефону якої введений з клавіатури; якщо такої людини немає, вивести на екран відповідне повідомлення.

## **Варіант 13.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людей, що народилися в тому місяці, значення якого введене з клавіатури; якщо таких немає, вивести на екран відповідне повідомлення.

## **Варіант 14.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людину, прізвище якої введене з клавіатури; якщо такої немає, вивести на екран відповідне повідомлення.

### Варіант 15.

Є колекція, що містить список друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людину, чиє прізвище введене з клавіатури; якщо такої людини немає, вивести на екран відповідне повідомлення.

### Варіант 16.

Є колекція, що містить список друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людей, що народилися під знаком, найменування якого введене з клавіатури; якщо таких немає, вивести на екран відповідне повідомлення.

### Варіант 17.

Є колекція, що містить список друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людей, що народилися в тому місяці, значення якого введене з клавіатури; якщо таких немає, вивести на екран відповідне повідомлення.

### Варіант 18.

Є колекція, що містить список товарів, кожний елемент містить наступні дані:



- назва товару;
- назва магазину, в якому продається товар;
- вартість товару в гривнях.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товар, назва якого введена з клавіатури; якщо таких товарів немає, вивести на екран відповідне повідомлення.

### **Варіант 19.**

Є колекція, що містить список товарів, кожний елемент містить наступні дані:

- назва товару;
- назва магазину, в якому продається товар;
- вартість товару в гривнях.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товари, що продаються в магазині, назва якого введена з клавіатури; якщо такого магазину немає, вивести на екран відповідне повідомлення.

### **Варіант 20.**

Є колекція, що містить список переказів, кожний елемент містить наступні дані:

- розрахунковий рахунок платника;
- розрахунковий рахунок одержувача;
- перерахована сума в гривнях.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про суму, зняту з розрахункового рахунку платника, введеного з клавіатури; якщо такого розрахункового рахунку немає, вивести на екран відповідне повідомлення.

### **Варіант 21.**

Є колекція, що містить список маршрутів, кожний елемент містить наступні дані:

- назву початкового пункту маршруту;
- назву кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про кількість маршрутів, які починаються або закінчуються в пункті, назва якого введена з клавіатури; якщо таких маршрутів немає, вивести на екран відповідне повідомлення.

## **Варіант 22.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- день народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про номер телефону людини, прізвище якої введене з клавіатури; якщо такої людини немає, вивести на екран відповідне повідомлення.

## **Варіант 23.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про кількість людей, що народилися в тому місяці, значення якого введене з клавіатури; якщо таких немає, вивести на екран відповідне повідомлення.

## **Варіант 24.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я (прізвища можуть повторюватися);
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;

- здійснює вивід на екран інформації про людей, що мають прізвище, введене з клавіатури; якщо таких немає, вивести на екран відповідне повідомлення.

### **Варіант 25.**

Є колекція, що містить список друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про кількість людей, чий знак Зодіаку введений з клавіатури; якщо таких людей немає, вивести на екран відповідне повідомлення.

### **Варіант 26.**

В колекції записана звітна відомість результатів екзаменаційної сесії студентської групи, яка для кожного студента містить прізвище, ініціали і оцінки з п'яти предметів.

Скласти програму, за допомогою якої можна створювати, переглядати та поповнювати колекцію і отримувати:

- список всіх студентів;
- список студентів, що склали іспити тільки на «5»;
- список студентів, що мають трійки;
- список студентів, що мають двійки. При цьому студент, що має більш ніж одну двійку, виключається із списку.

### **Варіант 27.**

Підприємство має місцеву телефонну станцію. В колекції записано телефонний довідник даного підприємства, який для кожного номера телефону містить номер приміщення і список службовців, що сидять в даному приміщенні.

Скласти програму, яка:

- створює, переглядає та поповнює базу;
- за номером телефону видає номер приміщення і список людей, що сидять в ньому;
- за номером приміщення видає номер телефону;
- за прізвищем службовця видає номер телефону і номер приміщення.

Номер телефону – двозначний. У одному приміщенні може знаходитися від одного до чотирьох службовців.

### **Варіант 28.**

У готелі є 15 номерів, з них 5 одномісних і 10 двомісних. Скласти програму, яка створює, переглядає та поповнює колекцію, що містить дані про мешканців і за прізвищем визначає номер, де проживає мешканець.

Програма запрошує прізвище мешканця.

- Якщо мешканця з таким прізвищем немає, про це видається повідомлення.
- Якщо мешканець з таким прізвищем в готелі єдиний, програма видає прізвище мешканця і номер помешкання.
- Якщо в готелі проживає два або більше мешканців з таким прізвищем, програма додатково запрошує ініціали.

### **Варіант 29.**

Є колекція, що містить список службовців. Для кожного службовця вказано прізвище і ініціали, назву посади, рік прийому на роботу і оклад (величину заробітної плати).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про службовця, прізвище якого введено з клавіатури.

### **Варіант 30.**

Розклад електричок зберігається у вигляді колекції. Кожен елемент містить назву пункту відправлення, назву пункту призначення, позначки типу «звичайний», «підвищеного комфорту», «швидкісний експрес» та час відправлення.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- виводить на екран інформації про поїзди, що відходять після введеного часу.
- виводить на екран інформації про поїзди, що відходять із заданого пункту відправлення.
- виводить на екран інформації про поїзди, що відходять до заданого пункту призначення.

### **Варіант 31.**

Є колекція, що містить список товарів. Для кожного товару вказані його назва, вартість одиниці товару в гривнях, кількість і одиниця вимірювання (наприклад, кількість: 100 шт., одиниця вимірювання: упаковка по 20 кг.).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товар, назва якого введена з клавіатури;
- здійснює вивід на екран інформації про товари із заданого з клавіатури діапазону вартості.

### **Варіант 32.**

Є колекція, що містить список товарів. Для кожного товару вказані його назва, назва магазину, в якому продається товар, вартість одиниці товару в гривнях і його кількість з вказівкою одиниці вимірювання (наприклад, кількість: 100 шт., одиниця вимірювання: упаковка по 20 кг.).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товар, назва якого введена з клавіатури;
- здійснює вивід на екран інформації про товари, що продаються в магазині, назва якого введена з клавіатури.

### **Варіант 33.**

Є колекція, що містить список студентської групи. Кожен елемент містить прізвище студента і три екзаменаційні оцінки, причому список ніяк не впорядкований.

Скласти програму, яка створює, переглядає та поповнює колекцію.

### **Варіант 34.**

Є колекція, що містить список товарів. Для кожного товару вказані його назва, назва магазину, в якому продається товар, вартість одиниці товару в гривнях і його кількість з вказівкою одиниці вимірювання (наприклад, кількість: 100 шт., одиниця вимірювання: упаковка по 20 кг.).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про товари, що продаються в магазині, назва якого введена з клавіатури;

- здійснює вивід на екран інформації про товари із заданого з клавіатури діапазону вартості.

### **Варіант 35.**

Є колекція, що містить список маршрутів, кожний елемент містить наступні дані:

- назву початкового пункту маршруту;
- назву кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про маршрут, номер якого введений з клавіатури; якщо такого маршруту немає, вивести на екран відповідне повідомлення.

### **Варіант 36.**

Є колекція, що містить список маршрутів, кожний елемент містить наступні дані:

- назву початкового пункту маршруту;
- назву кінцевого пункту маршруту;
- номер маршруту.

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про маршрути, які починаються або закінчуються в пункті, назва якого введена з клавіатури; якщо таких маршрутів немає, вивести на екран відповідне повідомлення.

### **Варіант 37.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людину, номер телефону якої введений з клавіатури; якщо такої людини немає, вивести на екран відповідне повідомлення.

### **Варіант 38.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людей, що народилися в тому місяці, значення якого введене з клавіатури; якщо таких немає, вивести на екран відповідне повідомлення.

### **Варіант 39.**

Є колекція, що містить список телефонів друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- номер телефону;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людину, прізвище якої введене з клавіатури; якщо такої немає, вивести на екран відповідне повідомлення.

### **Варіант 40.**

Є колекція, що містить список друзів, кожний елемент містить наступні дані:

- прізвище, ім'я;
- знак Зодіаку;
- дата народження (масив з трьох чисел).

Написати програму, що виконує наступні дії:

- створює, переглядає та поповнює колекцію;
- здійснює вивід на екран інформації про людину, чиє прізвище введене з клавіатури; якщо такої людини немає, вивести на екран відповідне повідомлення.

# ***Питання та завдання для контролю знань***

## **Контейнери**

1. Що таке контейнер?
2. Які види контейнерів STL Ви знаєте?
3. Які види операцій над контейнерами STL Ви знаєте?
4. Які способи доступу до елементів контейнера Вам відомі?
5. Чим відрізняється прямий доступ від послідовного?
6. Чим відрізняється прямий доступ від асоціативного?
7. Чим відрізняється послідовний доступ від асоціативного?
8. Перелічіть операції, які реалізуються зазвичай при послідовному доступі до елементів контейнера.
9. Перелічіть операції, які реалізуються зазвичай для об'єднання контейнерів.
10. Які Ви знаєте контейнери послідовного доступу STL?
11. Які Ви знаєте асоціативні контейнери STL?
12. Які Ви знаєте адаптери контейнерів STL?
13. Що таке стек?
14. Що таке черга?
15. Що таке дек?

## **Ітератори**

1. Що таке ітератор? (поясніть поняття та дайте визначення)
2. Що служить ітератором для вбудованого масиву?
3. Що служить ітератором для зв'язаного списку?
4. Чи можна реалізувати послідовний доступ без ітератора?
5. В чому перевага реалізації послідовного доступу за допомогою ітератора?
6. Які Ви знаєте категорії ітераторів?
7. Які категорії ітераторів мають найбільше можливостей?
8. Які категорії ітераторів мають найбільше можливостей?
9. Які операції можна застосовувати до будь-яких ітераторів?
10. Призначення методів `begin()` та `end()`

## **Алгоритми**

1. Що таке алгоритм STL?
2. Які види алгоритмів STL Ви знаєте?
3. Які параметри зазвичай приймають функції, що реалізують алгоритми STL?



4. Які Ви знаєте алгоритми, що не модифікують послідовність?
5. Які Ви знаєте алгоритми, що модифікують послідовність?
6. Які Ви знаєте алгоритми, що опрацьовують розбиття послідовності?
7. Які Ви знаєте алгоритми сортування?
8. Які Ви знаєте алгоритми бінарного пошуку?
9. Які Ви знаєте алгоритми злиття послідовностей?
10. Які Ви знаєте алгоритми, що опрацьовують динамічну пам'ять?
11. Які Ви знаєте алгоритми пошуку мінімального / максимального?
12. Які Ви знаєте алгоритми, що реалізують інші операції?

# Предметний покажчик

## A

adjacent\_find, 87  
advance, 64  
all\_of, 81  
any\_of, 81  
array, 36  
auto, 23

## B

back\_insert\_iterator, 75  
back\_inserter, 68  
begin, 65  
binary\_search, 132

## C

copy, 95  
copy\_backward, 97  
copy\_if, 97  
copy\_n, 96  
count, 88  
count\_if, 89

## D

deque, 38  
distance, 65

## E

end, 65  
equal, 90  
equal\_range, 131  
erase, 41

## F

fill, 105  
fill\_n, 106  
find, 83  
find\_end, 85  
find\_first\_of, 86  
find\_if, 84  
find\_if\_not, 84  
for – цикл на основі діапазону, 25  
for\_each, 82  
forward\_list, 39  
front\_insert\_iterator, 75  
front\_inserter, 69

## G

generate, 106  
generate\_n, 107

## I

includes, 136  
inplace\_merge, 135

insert, 42  
insert\_iterator, 76  
inserter, 69  
is\_heap, 145  
is\_heap\_until, 146  
is\_partitioned, 118  
is\_permutation, 91  
is\_sorted, 127  
is\_sorted\_until, 127  
istream\_iterator, 77  
istreambuf\_iterator, 78  
iter\_swap, 101  
iterator, 71  
iterator\_traits, 72

## L

lexicographical\_compare, 152  
list, 39  
lower\_bound, 130

## M

make\_heap, 143  
make\_move\_iterator, 70  
map, 43  
max, 147  
max\_element, 149  
merge, 134  
min, 147  
min\_element, 148  
minmax, 148  
minmax\_element, 150  
mismatch, 89  
move, 98  
move\_backward, 99  
move\_iterator, 74

## N

next, 67  
next\_permutation, 153  
none\_of, 82  
nth\_element, 128

## O

ostream\_iterator, 77  
ostreambuf\_iterator, 79

## P

partial\_sort, 125  
partial\_sort\_copy, 126  
partition, 119  
partition\_copy, 120  
partition\_point, 121  
pop\_heap, 143  
prev, 67  
prev\_permutation, 154  
priority\_queue, 45  
push\_heap, 142

## Q

queue, 45

## R

random\_shuffle, 115  
rbegin, 66  
remove, 108  
remove\_copy, 109  
remove\_copy\_if, 110  
remove\_if, 109  
rend, 66  
replace, 103  
replace\_copy, 104  
replace\_copy\_if, 105  
replace\_if, 103  
reverse, 113  
reverse\_copy, 113  
reverse\_iterator, 73  
rotate, 114  
rotate\_copy, 114

## S

search, 92  
search\_n, 93  
set, 41  
    erase, 41  
    insert, 42  
set\_difference, 138  
set\_intersection, 137  
set\_symmetric\_difference, 140  
set\_union, 136  
shuffle, 116  
sort, 123  
sort\_heap, 144  
stable\_partition, 120  
stable\_sort, 124  
stack, 45  
swap, 100  
swap\_ranges, 100

## T

transform, 102

## U

unique, 111  
unique\_copy, 112  
upper\_bound, 130

## V

vector, 37

## A

Адаптер ітератора, 19, 59  
Адаптер контейнера, 45  
Алгоритм, 16, 32  
Алгоритми, 80  
    min / max, 147  
    бінарний пошук, 130  
    злиття, 134  
    купа, 142  
    лексикографічні, 152  
    модифікуючі, 95  
    не модифікуючі, 80  
    розбиття, 118  
    сортування, 123  
Асоціативний контейнер, 16, 32, 33, 41

## B

Втрата ітератора, 47

## I

Ітератор, 16, 18, 32, 55

## K

Категорії ітераторів, 55  
Контейнер, 16, 32

## L

Лямбда-вираз, 27

## H

Невпорядкований асоціативний контейнер, 32, 44

## P

Послідовний контейнер, 16, 32, 34

## Φ

Функтор, 16, 32

# Література

## Основна

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня СПб.: Питер, 2007. – 461 с.
2. Павловская Т.А., Щупак Ю.А. С/С++. Объектно-ориентированное программирование: Практикум СПб.: Питер, 2005. – 265 с.
3. Дейтел Х.М., Дейтел П.Дж. Как программировать на С++ М.: Бином-Пресс, 2005. – 1248 с.
4. Уэллин С. Как не надо программировать на С++ СПб.: Питер, 2004. – 240 с.
5. Хортон А. Visual C++ 2005: Базовый курс М.: Вильямс, 2007. – 1152 с.
6. Солтер Н.А., Клеппер С.Дж. С++ для профессионалов. М.: Вильямс, 2006. – 912 с.
7. Лафоре Р. Объектно-ориентированное программирование в С++ СПб.: Питер, 2006. – 928 с.
8. Лаптев В.В. С++. Объектно-ориентированное программирование СПб.: Питер, 2008. – 464 с.
9. Лаптев В.В., Морозов А.В., Бокова А.В. С++. Объектно-ориентированное программирование. Задачи и упражнения СПб.: Питер. – 2007. – 288 с.: ил.
10. <https://en.cppreference.com/w/cpp/container>
11. <https://en.cppreference.com/w/cpp/iterator>
12. <https://en.cppreference.com/w/cpp/algorithm>
13. <http://www.cplusplus.com/reference/iterator/>
14. <http://www.cplusplus.com/reference/algorithm/>

## Додаткова

15. Прата С. Язык программирования С++. Лекции и упражнения СПб.: ДиаСофт, 2003. – 1104 с.
16. Мейн М., Савитч У. Структуры данных и другие объекты в С++ М.: Вильямс, 2002. – 832 с.
17. Саттер Г. Решение сложных задач на С++ М.: Вильямс, 2003. – 400 с.
18. Чепмен Д. Освой самостоятельно Visual C++ .NET за 21 день М.: Вильямс, 2002. – 720 с.
19. Мартынов Н.Н. Программирование для Windows на С/С++ М.: Бином-Пресс, 2004. – 528 с.
20. Паппас К., Мюррей У. Эффективная работа: Visual C++ .NET СПб.: Питер, 2002. –

816 с. М.: Вильямс, 2001. – 832 с.

21. Грэхем И. Объектно-ориентированные методы. Принципы и практика М.: Вильямс, 2004. – 880 с.
22. Элиенс А. Принципы объектно-ориентированной разработки программ М.: Вильямс, 2002. – 496 с.
23. Ларман К. Применение UML и шаблонов проектирования М.: Вильямс, 2002. – 624 с.
24. Шилэт Г. Полный справочник по С. 4-е издание. М.-СПб.-К: Вильямс, 2002.
25. Прата С. Язык программирования С. Лекции и упражнения. М.: ДиаСофтЮП, 2002.
26. Александреску А. Современное проектирование на С++ М.: Вильямс, 2002.
27. Браунси К. Основные концепции структур данных и реализация в С++ М.: Вильямс, 2002.
28. Подбельский В.В. Язык СИ++. Учебное пособие. М.: Финансы и статистика, 2003.
29. Павловская Т.А., Щупак Ю.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2002.
30. Савитч У. Язык С++. Объектно-ориентированного программирования. М.-СПб.-К.: Вильямс, 2001.
31. <https://docs.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-160>
32. <https://docs.microsoft.com/en-us/cpp/cpp/range-based-for-statement-cpp?view=msvc-160>
33. <https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-160>