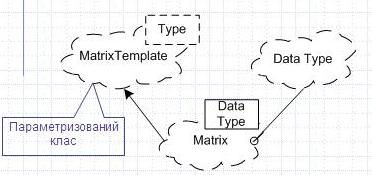
**Лекція №28. Класи колекцій стандартної бібліотеки**

**Відношення інстанціювання в нотації Буча**



**Загальна структура Стандартної бібліотеки С++ – STL**

*Стандартна бібліотека С++* являє собою колекцію класів і функцій для розв'язання складних і низькорівневих задач програмування. Бібліотека включає в себе такі компоненти:

* засоби для роботи з потоками введення / виведення;
* набір структурованих даних і алгоритмів, раніше відомих як Стандартна бібліотека шаблонів (Standard Template Library, STL);
* засоби локалізації (адаптації до національних мов);
* параметризований клас string;
* параметризований клас complex для подання комплексних величин;
* клас vallaray, оптимізований для обробки числових масивів;
* параметризований клас numeric\_limits і спеціалізації для кожного базового типу даних;
* засоби управління пам'яттю;
* велика підтримка національних наборів символів;
* засоби обробки винятків.

Засоби Стандартної бібліотеки визначено в просторі імен **std** і подано великим набором заголовних файлів.

С++ - це конгломерат мов програмування, структуру якого можна надати такою схемою:

Для ефективного використання С++ треба знати і правильно використовувати особливості та правила кожної підмови.

STL − найбільш потужна частина мови; STL є частиною мови (стандарт С++11). Дозволяє не “придумувати велосипеда”. Структуру STL можна надати такою схемою.

Одна з найбільш незвичайних ідей в STL − це *узагальнені алгоритми*. Поведінка стандартних контейнерів є мінімальною, однак вони призначені для спільного функціонування з набором узагальнених алгоритмів.

**Основні будівельні блоки STL**

До основних будівельних блоків, що надаються STL, можна віднести контейнери, ітератори й алгоритми. ***Контейнер*** – це клас, який зберігає колекцію інших об'єктів і включає базові функції для підтримки використання загальних алгоритмів. Стандартні контейнери не є похідними від деякого загального базового класу. Замість цього кожен контейнер забезпечує набір стандартних операцій зі стандартними іменами і сенсом.

Є такі основні групи контейнерів:

* послідовності – вектор (vector), список (list), дек (deque, черга з двома кінцями);
* адаптери послідовностей – стек (stack), черга (queue), черга з пріоритетом (priority\_queue);
* асоціативні контейнери – асоціативні масиви (map, multimap) і множини (set, multiset).

У спеціальну групу входять класи, побудовані на стандартних контейнерах і спеціалізовані для конкретного використання – рядки (string), масиви значень (valarray) і бітові набори (bitset).

Найчастіше контейнери ідентифікуються в однойменних заголовних файлах. Клас multimap визначено в заголовному файлі <map>. Клас multiset визначено в заголовному файлі <set>.

Прохід (ітерація) по контейнеру здійснюється шляхом визначення класу ітератора, придатного для певного виду контейнера. ***Ітератор*** – це об'єкт, який абстрагує поняття вказівника на елемент послідовності і дозволяє обходити елементи послідовності в певному напрямку. Кожен контейнерний клас в Стандартній бібліотеці С++ здатний згенерувати ітератор, який реалізує оптимальні механізми проходження елементів контейнера.

Об'єкт-ітератор повинен підтримувати такі операції:

* отримання поточного елементу (реалізовано операторами \* і ->);
* інкремент (реалізовано оператором ++);
* перевірка на рівність (реалізовано оператором ==);

Розрізняють *односпрямовані ітератори* (для запису і для читання), *двоспрямовані ітератори*, *ітератори з довільним доступом*. Вони відрізняються кількістю визначених для них операцій. Для двоспрямованого ітератора додається операція --, для ітераторів з довільним доступом – операції +, -, +=, -=, <, >, <=, >=, а також [].

***Алгоритми*** – це шаблони функцій (набори шаблонів функцій), які працюють з послідовностями через їх ітератори. Оголошення більшості стандартних алгоритмів знаходяться в заголовному файлі <algorithm>. Декілька узагальнених чисельних алгоритмів визначено в заголовному файлі <numeric>.

**Контейнери Стандартної бібліотеки шаблонів**

**Стандартні послідовні контейнери**

У Стандартній бібліотеці визначені два типи контейнерів – послідовності й асоціативні контейнери. До ***послідовностей*** відносяться такі контейнери, як vector (вектор), list (список), deque (дек, черга з двома кінцями).

Для всіх послідовностей визначені такі типи:

* value\_type – тип елемента,
* allocator\_type – тип розподільника пам'яті,
* size\_type – тип індексу, лічильників і т.д. (зазвичайunsigned int),
* difference\_type – тип різниці між ітераторами,
* reference і const\_reference – посилання на елемент.

Визначено також типи стандартних ітераторів: iterator, reverse\_iterator, const\_iterator, const\_reverse\_iterator.

Для отримання ітератора першого елемента і елемента, наступного за останнім відповідно в прямій і зворотній послідовності використовуються відповідно функції-елементи begin(), end(), rbegin() і rend(). Крім використання ітераторів, доступ до першого і останнього елементів може бути здійснений за допомогою функцій-елементів front() і back().

Функція-елемент size() повертає число елементів, empty() перевіряє, чи порожній контейнер, resize(size\_type) змінює розмір контейнера, swap(container) міняє місцями елементи двох контейнерів.

Окрім усталеного конструктора і конструктора копіювання, у послідовностей також є конструктор, що задає початковий розмір, конструктор, що задає початковий розмір і значення, яким заповнюється контейнер, а також конструктор, що заповнює елементи значеннями з іншої послідовності за допомогою двох ітераторів.

***Вектор* (**vector) багато в чому аналогічний традиційному одновимірному масиву. Для використання векторів до вихідного файлу треба підключити заголовний файл <vector>. Тип елемента вказується в кутових дужках ("<" та ">"). Під час ініціалізації вектора можна визначати його розмірність. У наведеному нижче прикладі визначається змінна a як вектор з n дійсних чисел:

#include <vector>

**using** std::vector;

vector<**double**> a(n);

Тут n може бути як константою, так і змінною.

Звертатися до окремих елементів можна за індексом, як до елементів масиву, наприклад:

**for** (**int** i = 0; і < a.size(); i++)

{

cіn >> a[i];

}

Можна також звертатися до елементів за допомогою функції at() (як для читання, так і для запису, оскільки ця функція повертає посилання):

**for** (**int** i = 0; і < a.size(); i++)

{

a.at(i) = a.at(i) + 9;

}

Відмінність між [] і at() полягає в тому, що функція at() генерує виняток, якщо індекс, вказаний як параметр, виходить за межі індексів.

Можна також описати "порожній" вектор (довжиною 0), а потім додавати елементи у кінець за допомогою функції-елементу push\_back(). В такий спосіб можна отримати вектор, елементи якого дорівнюють 0, 1, 2, 3 та 4:

vector<**int**> b;

**for** (**int** і = 0; і < 5; і++)

{

b.push\_back(i);

}

За допомогою функції pop\_back() можна видалити останній елемент. Функції push\_back() і pop\_back() можуть бути застосовані до всіх послідовних контейнерів.

Вектор може містити в собі вектори. Таким чином можна імітувати двовимірний масив.

Контейнер vector може надавати такі ітератори:

* vector<T>::iterator – ітератор, який реалізує прямий прохід по контейнеру;
* vector<T>::reverse\_iterator – ітератор, який реалізує зворотний прохід по контейнеру;
* vector<T>::const\_iterator – ітератор, через який не можна змінювати елементи контейнера;
* vector<T>::const\_reverse\_iterator – константний ітератор, який реалізує зворотний прохід по контейнеру.

Ітератори вектора є ітераторами довільного доступу (не тільки послідовними). Для них реалізовані операції складання з цілими і віднімання цілих.

Безпосередньо як об'єкт ітератор всередині контейнера не міститься, там описаний тільки його тип. Змінну-ітератор треба визначати окремо:

vector<**int**>::iterator vi;

// v.begin() і v.end() повертають ітератори,

// які вказують на початок і за кінець вектора

// Змінюємо значення через ітератор:

**for** (vi = v.begin(); vi != v.end(); vi++)

{

\*vi = 200;

}

За допомогою ітератора reverse\_iterator можна обійти вектор з кінця до початку:

vector<**int**>::reverse\_iterator ri;

// v.rbegin() повертає ітератор для зворотного проходу,

// який вказує на кінець вектору

// v.rend() вказує на позицію перед початком вектору

// ri++ просуває ітератор на попередній елемент

**for** (ri = v.rbegin(); ri != v.rend(); ri++)

{

cout << \*ri; // прохід від кінця до початку

}

Ітератор const\_iterator працює аналогічно звичайному, за винятком того, що через нього не можна міняти елементи масиву. Функція-елемент

**insert(iterator pos, const T& x);**

вставляє х перед позицією pos. Функція-елемент

**erase(iterator first, iterator last);**

видаляє підпослідовність з вектору.

***Список*** (list) – це послідовність, оптимізована для вставки і видалення елементів. Цей контейнер реалізований класичним двозв'язним списком. Найбільш типові операції для роботи зі списком:

* insert(p, x) – додавання х перед p;
* insert(p, n, x) – додавання n копій х перед p;
* insert(p, first, last) – додавання перед p елементів послідовності, заданої ітераторами first і last;
* erase(p) – видалення елементу в позиції p;
* erase(first, last) – видалення послідовності, заданої ітераторами;
* clear() – видалення всіх елементів.

Наведений нижче приклад демонструє використання деяких функцій-елементів класу list:

std::list<**int**> list = { 1, 2, 4, 8, 16 }; // ініціалізація списком значень доступна у версії C++11

list.insert(++++list.begin(), 3); // 1, 2, 3, 4, 8, 16

list.insert(--list.end(), 2, 12); // 1, 2, 3, 4, 8, 12, 12, 16

list.erase(--list.end()); // 1, 2, 3, 4, 8, 12, 12

Клас "**Черга з двома кінцями**" (**deque**) схожий на vector, але з можливістю вставки і видалення елементів на обох кінцях. Клас реалізовано достатньо ефективно. Розглянемо приклад:

std::deque<**int**> deque = { 1, 2, 4, 8, 16 };

std::cout << deque[0] << std::endl; // 1

deque.push\_front(0);

std::cout << deque[0] << std::endl; // 0

deque.push\_back(32); // 0, 1, 2, 4, 8, 16, 32

deque.pop\_front();

std::cout << deque[0] << std::endl; // 1

deque.pop\_front();

std::cout << deque[0] << std::endl; // 2

deque.pop\_back();

deque.pop\_back(); // 2, 4, 8

Клас deque використаний, зокрема, для внутрішньої реалізації адаптерів послідовностей (будуть розглянуті нижче).

**Рядки**

Для роботи з послідовностями символів (рядків) Стандартна бібліотека С++ пропонує спеціальний тип – std::string. Він визначений за допомогою **typedef** як інстанційований клас шаблонного класу basic\_string:

**typedef** basic\_string<**char**> string;

Шаблон basic\_string – це спеціалізований вектор, оптимізований для зберігання рядків. Цей шаблон можна використовувати для створення конкретних типів рядків, які відрізняються представленням символів. Зокрема, тип wstring працює з 16-бітовими символами.

Для роботи з рядками необхідно підключити заголовний файл <string>. Конструктор без параметрів створює порожній рядок. Рядок також можна ініціювати ланцюжком символів або іншим рядком типу std::string:

string s1;

string s2("a string");

string s3 = "initial value";

string s4(s3);

string s5 = s4;

Загальна кількість символів повертається функцією-елементом **length()**, який є синонімом функції size(). Можна звертатись до окремих символів за індексом або через функцію at(). Наприклад, усі символи рядка s у наступному прикладі замінюються символом 'a':

**for** (**int** i = 0; i < s.length(); i++)

{

s[i] = 'a';

}

Клас надає низку корисних функцій-елементів для пошуку, заміни, вставки, обміну вмісту тощо. Зокрема, за допомогою функції-елементу **resize()** змінюється поточна довжина рядка. Другим параметром функції можна вказати символ, що вставляється в нові позиції:

s7.resize(15, '\t'); // вставка символів табуляції

Функція-елемент **empty()** повертає **true**, якщо рядок не містить символів. Оператор + здійснює зшивання двох рядків:

s = s1 + s2;

Можна також застосовувати +=. Рядку може бути присвоєно значення іншого рядка, масиву символів, або окремого символу:

s1 = s2;

s2 = "a new value";

s3 = 'x';

Оператор +=, як і оператор + з відповідним другим аргументом, може бути використаний для всіх трьох форм присвоєння.

Функція-елемент **c\_str()** повертає тимчасовий вказівник на масив символів що закінчується нулем. Наприклад:

string s = "Text";

cout << s.length() << endl; // 4

cout << strlen(s.c\_str()) << endl; // те ж саме

*Примітка*. У наведеному вище прикладі використання c\_str() не має особливого сенсу. Сенс виникає, коли відповідна функція з параметром типу string відсутня.

Для використання стандартних алгоритмів можна працювати з ітеаторами довільного доступу і відповідними функціями **begin()** і **end()**. Зворотні ітератори отримують за допомогою функцій **rbegin()** і **rend()**.

Функції-елементи **insert()** і **erase()** аналогічні відповідним функціям вектору. Для їх роботи необхідно визначити як параметри відповідні ітератори. Функція-елемент **replace()** – комбінація функцій erase() і insert():

s2.replace(s2.begin()+3, s2.begin()+6, s3.begin(), s3.end());

Альтернативною реалізацією функцій є використання цілих позицій і рядкових констант:

s3.insert(3, "abc"); // вставка abc після позиції 3

s3.erase(4, 2); // видалення позицій 4 і 5

s3.replace(4, 2, "pqr"); // заміна позицій 4 і 5 на pqr

Функція-елемент **copy()** копіює в рядок символи з зазначеного діапазону:

s3.copy(s4, 2); // присвоює s4 позиції від 2 до кінця s3

s5.copy(s4, 2, 3); // присвоює s4 позиції від 2 до 4 рядка s5

Функція-елемент **substr()** повертає рядок, що є частиною вихідного. При цьому також задається діапазон індексів:

cout << s4.substr(3) << endl ; // виведення від позиції 3 до кінця

cout << s4.substr(3, 2) << endl ; // виведення позицій 3 і 4

Функцію-елемент **compare()** використовують для лексикографічного порівняння рядків. Необов'язкові аргументи можуть задавати різні вихідні позиції для порівняння. Функція повертає від'ємне значення, якщо одержувач менше аргументу, нуль, якщо вони рівні, і додатне в іншому випадку. Частіше використовуються операції порівняння <, <=, ==, !=, >= і >, які також можуть застосовуватися для порівняння з ланцюжком символів.

Функція-елемент **find()** визначає перше включення аргументу в поточний рядок. Необов'язковий цілий аргумент визначає стартову позицію для пошуку. Функція повертає знайдену позицію або значення за межами індексу. Функція **rfind()** шукає з кінця в зворотному напрямку.

string s1 = "mississippi";

cout << s1.find("ss") << endl; // повертає 2

cout << s1.find("ss", 3) << endl; // повертає 5

cout << s1.rfind("ss") << endl; // повертає 5

cout << s1.rfind("ss", 4) << endl; // повертає 2

**Адаптери послідовностей**

Класи **stack** (стек), **queue** (черга) та **priority\_queue** (черга з пріоритетом) визначено не як окремі контейнери, а як адаптери базових контейнерів. Адаптери контейнерів надають обмежений інтерфейс до контейнера. Зокрема, вони не надають ітераторів. Стандартні контейнери зберігаються як елементи даних у класах-адаптерах в розділі **private**.

Адаптер stack визначено в заголовному файлі <stack>. Стек – це динамічна структура даних, що складається з змінного числа елементів. Додавання нових і видалення елементів здійснюється з одного кінця стека за принципом LIFO (Last In – First Out – останнім увійшов – першим вийшов). Стандартна реалізація передбачає збереження елементів у черзі з двома кінцями, але можна використовувати будь-яку послідовність, задавши її другим параметром шаблону. Для роботи зі стеком використовують функції **top**() (отримати елемент з вершини), **push**() (додати елемент) і pop() (видалити елемент).

Клас queue визначено в заголовному файлі <queue>. Допускається операція push() (додати елемент в кінець черги), pop() (видалити елемент з початку черги), **front()** і **back()**, які отримують елементи з початку і кінця черги. В обох класів реалізовані функції **empty()** і **size().**

У наведеному нижче прикладі створюється черга цілих чисел на базі списку (list):

#include <iostream>

#include <queue>

#include <list>

**using** std::cout;

**using** std::endl;

**using** std::list;

**using** std::queue;

**int** main()

{

// Замість deque можна використовувати list:

queue<**int**, list<**int**> > q;

q.push(2);

q.push(3);

// Можна отримати значення останнього елементу:

cout << q.back() << endl;

// Отримаємо елементи з черги:

**while** (q.size()) // або while (!q.empty())

{

cout << q.front() << endl;

q.pop();

}

**return** 0;

}

Для представлення черг і списків не можна використовувати vector, оскільки він не реалізує функції pop\_front().

У Стандартної бібліотеці С++ визначено адаптер послідовності "Черга з пріоритетом" (priority\_queue). Кожному елементу такої черги призначений пріоритет, який визначає порядок, в якому елементи виходять з черги. Оголошення priority\_queue знаходиться в заголовному файлі <queue>. В усталеному випадку елементи порівнюються за допомогою оператора < і функція top() повертає найбільший елемент.

Функція **size()** повертає кількість елементів в черзі. Функція **empty()** з результуючим значенням типу **bool** повертає **true**, якщо чергу порожня. Функція **pop()** видаляє позицію з найвищим пріоритетом з черги. За допомогою спеціального конструктора можна проініціалізувати чергу з пріоритетом, задавши два ітератори з існуючої послідовності. Замість використання оператора < для організації порівняння можна використовувати клас, що містить функцію елемент **bool** **operator()()**, приймаючу два аргументи типу константного посилання на тип елементу черги.

Черги з пріоритетами найчастіше використовуються для моделювання черги подій і їх обробки. Для представлення події можна використовувати структуру або клас, для якого перевантажена операція <.

**Асоціативні масиви та множини**

*Асоціативний масив* (map, його іноді називають словником) зберігає пари значень. Маючи одне зі значень (ключ), можна отримати доступ до іншого (значення). Асоціативний масив можна представляти як масив, в якому індекс не обов'язково є цілим.

Для реалізації такого масиву в Стандартній бібліотеці є два класи: map і multimap. Вони відрізняються тим, що в контейнері map ключі повинні бути унікальними, а в контейнері multimap – ні (ключі можуть повторюватися).

Для роботи з map і multimap необхідно підключати заголовний файл <map>.

Клас map є параметризованим класом, двома параметрами якого є тип ключа і тип значення. Для створення елементу масиву досить присвоїти елементу нове значення, при чому індексом є ключ. Якщо ключ не знайдений, то в масив поміщається пара "ключ" – "усталене значення для типу значення". Наприклад, якщо ключ – рядок, а значення – ціле, то можна створити таку змінну:

map<string, **int**> m;

Новий елемент включається шляхом присвоєння:

m["string1"] = 7;

Прочитати значення можна аналогічно:

**int** i = m["string1"];

Ця операція завжди дає якийсь коректний результат. Повертається або введене значення (якщо ключ знайдений), або усталене значення для типу значення.

Об'єкт класу map містить об'єкти структури pair, яка складається з двох елементів – first і second. За допомогою функції-елементу insert і визначеної в Стандартній бібліотеці функції make\_pair можна вставити нову пару в асоціативний масив без використання операції []:

m.insert(make\_pair(string("string1"),8));

За допомогою функції count() можна визначити, скільки разів в асоціативний масив входить пара з заданим ключем (у разі використання map count() поверне 0 або 1).

**if** (m.count("string1") != 0) // елемент з цим ключем є в масиві

Функція-елемент **find()** дозволяє отримати значення по заданому ключу. Вона повертає ітератор, який вказує на знайдений елемент (об'єкт типу pair), якщо він є або який дорівнює ітератору, який повертає end(), якщо елемент із заданим ключем не знайдено. Вставка нових елементів при цьому не проводиться (на відміну від операції []).

map<string,**int**>::iterator mi;

**if** ((mi = m.find("key1")) != m.end())

{

cout << "значення=" << mi->second << endl;

}

Всі елементи асоціативного масиву можна обійти за допомогою ітератора:

map<string,**int**>::iterator mi;

**for** (mi = m.begin(); mi != m.end(); mi++)

{

cout << "key=" << mi->first << " value=" << mi->second << endl;

}

Елементи зберігаються в асоціативному масиві в порядку зростання ключа. Зазвичай використовується операція <. Замість використання оператора < для організації порівняння можна використовувати клас, що містить функцію елемент **bool operator**()(), яка приймає два аргументи типу константного посилання на тип ключа.

За допомогою функції-елементу **erase()** можна видалити задані елементи. Є три варіанти видалення:

m.erase(4); // За значенням ключа

m.erase(iter); // З використанням ітератора

m.erase(iter1, iter2); // З використанням пари ітераторів

На відміну від класу map, в об'єктах класу multimap значення ключа може повторюватися. Отже, втрачає сенс операція індексації [] і для multimap вона не визначена. Для отримання всіх елементів з певним ключем застосовуються функції-елементи equal\_range(), lower\_bound() і upper\_bound(). Перша з цих функцій приймає значення ключа і повертає пару (pair) з двох ітераторів, перший з яких вказує на перший елемент із заданим ключем, а другий – після останнього елементу. Ці ітератори можна отримати окремо за допомогою функцій lower\_bound() і upper\_bound():

multimap<string, **int**> mm;

**typedef** multimap<string, **int**>::iterator MmIter;

pair<MmIter, MmIter> p = mm.equal\_range("abc");

**for** (MmIter mmi = p.first; mmi != p.second; mmi++)

// Обхід елементів з однаковим ключем

**for** (MmIter mmi = mm.lower\_bound("abc"); mmi != mm.upper\_bound("abc"); mmi++)

// Теж саме

Вставка пар здійснюється за допомогою функції-елементу insert(). Перевірка наявності елементів із заданими ключами здійснюється за допомогою функції count(), яка може повернути значення, більше одиниці.

***Множину*** можна розглядати як варіант асоціативного масиву, в якому присутні тільки ключі. Елементи множини автоматично сортуються. Для множин не реалізована операція індексації. Найбільш вживані операції з множинами – додати елемент (insert()), видалити елемент (erase()), перевірити наявність елементу (find(), count()).

Звичайна множина (тип set) може містити тільки унікальні елементи. Спеціальний тип multiset допускає повторення елементів множини. Для роботи з обома множинами слід підключати заголовний файл <set>. Під час визначення змінних типу множини необхідно вказувати тільки один тип – тип елементу множини, наприклад:

set<string> words;

Додавання елементів здійснюється за допомогою функції-елементу insert():

words.insert("word1");

Визначення числа входжень елемента і пошук елемента аналогічні map. Визначення кількості входження:

**if** (words.count("word1") != 0)

{

// елемент у множині є

}

Пошук:

set<string>::iterator si;

**if** ((si = words.find("word1")) != words.end())

{

// цей елемент є в множині

cout << "елемент=" << \*si << endl;

}

Обхід елементів множини здійснюється за допомогою ітераторів:

// виведення множини

**for** (si = words.begin(); si != words.end(); si++)

{

cout << \*si << " ";

}

**Алгоритми Стандартної бібліотеки**

**Загальний огляд алгоритмів**

***Алгоритм*** (algorithm) – це шаблонна функція, оголошена в просторі імен std і в заголовному файлі <algorithm>, яка працює з елементами довільних послідовностей, заданих ітераторами і визначає обчислювальну процедуру.

Усі алгоритми відокремлені від деталей реалізації структур даних і використовують як параметри типи ітераторів. Тому вони можуть працювати зі створеними користувачем структурами даних, якщо ці структури даних мають типи ітераторів, що задовольняють припущеннями в алгоритмах. Шаблонні алгоритми Стандартної бібліотеки працюють не тільки зі структурами даних в бібліотеці, але також і з масивами C++.

Слід пам'ятати, що результат застосування алгоритмів до неприпустимих діапазонів не визначений. Немає ніяких механізмів перевірки коректності діапазону під час роботи програми. За коректність діапазону відповідає програміст. Існує кілька груп алгоритмів:

| **Групи алгоритмів** | **Приклади;** |
| --- | --- |
| алгоритми, які не модифікують послідовності | for\_earch(), find(), find\_if(), count() |
| алгоритми, що модифікують послідовність | transform(), copy(), replace(), fill(), swap(), random\_shuffle() |
| алгоритми сортування | sort(), partial\_sort(), stable\_sort() |
| алгоритми для роботи з множинами | includes(), set\_union(), set\_intersection() |
| алгоритми знаходження максимумів і мінімумів | min(), max(), min\_element(), max\_element() |
| алгоритми перестановок. | next\_permutation(), prev\_permutation() |

**Використання for\_each**

Розгляд загальних властивостей алгоритмів і використання допоміжних об'єктів можна почати з простого алгоритму, що не змінює послідовності: for\_each(). За допомогою цього алгоритму задаються операції з кожним елементом послідовності.

Алгоритм for\_each(InputIterator first, InputIterator last, Function f) застосовує f до результату розіменування кожного ітератора в діапазоні [first, last). Якщо f повертає результат, результат ігнорується.

У більшості випадків замість for\_each() доцільно використовувати інші алгоритми. Однак як приклад можна вивести на екран значення цілих за допомогою алгоритму for\_each():

#include <iostream>

#include <algorithm>

#include <vector>

**using** std::cout;

**using** std::endl;

**using** std::vector;

**using** std::for\_each;

**void** writeInt(**const** **int**& i)

{

cout << i << endl;

}

**int** main()

{

vector<**int**> a(6);

. . .

// формування вектора

. . .

// виведеня вектора:

for\_each(a.begin(), a.end(), writeInt);

**return** 0;

}

Якщо за допомогою функції for\_each() ми хочемо здійснити певні дії над масивом, замість ітератора слід використати вказівник на масив і на умовний елемент після масиву. Наприклад:

#include <iostream>

#include <algorithm>

**using** std::cout;

**using** std::endl;

**using** std::for\_each;

**void** writeInt(**const** **int**& i)

{

cout << i << endl;

}

**int** main()

{

**const** **int** n = 6;

**int** a[] = { 1, 2, 4, 8, 16, 32};

// виведеня масиву:

for\_each(a, a + n, writeInt);

**return** 0;

}

**Приклади використання transform**

Aлгоритм transform() приймає дві послідовності – вхідну (визначену двома ітераторами) і результуючу (визначену одним ітератором) і заповнює другу послідовність виконує над елементами першої послідовності дію, визначену функцією (останній параметр) і заповнює другу послідовність. У деяких випадках така функція може бути стандартною. В наведеному прикладі для кожного з рядків, які зберігаються в масиві, можна отримати довжину і записати в масив цілих:

#include <cstring>

#include <algorithm>

**using** std::transform;

**using** std::strlen;

**void** main()

{

**const** **char**\* words[] = { "first", "second", "third" };

**int** sizes[3];

transform(words, words + 3, sizes, strlen); // 5, 6, 5

// ... використання масиву sizes

}

Якщо замість масиву вказівників на символ створити масив рядків типу std::string, для алгоритму transform() доведеться створювати окрему функцію, оскільки як аргумент не можна використовувати функцію-елемент length():

#include <string>

#include <algorithm>

**using** std::transform;

**using** std::string;

**int** len(string s)

{

**return** s.length();

}

**void** main()

{

string words[] = { "first", "second", "third" };

**int** sizes[3];

transform(words, words + 3, sizes, len); // 5, 6, 5

// ... використання масиву sizes

}

Робота з векторами передбачає використання ітераторів замість вказівників. Можна заздалегідь створити порожній вектор необхідної довжини, але можна також скористатися функцією back\_inserter(), яка в свою чергу повертає спеціальний об'єкт-ітератор – back\_insert\_iterator, робота якого полягає в додаванні елементів в кінець контейнеру за допомогою функції push\_back(). Попередній приклад можна реалізувати за допомогою векторів:

#include <vector>

#include <string>

#include <algorithm>

**using** std::transform;

**using** std::string;

**using** std::vector;

**using** std::back\_inserter;

**int** len(string s)

{

**return** s.length();

}

**int** main()

{

vector<string> words = { "first", "second", "third" };

vector<**int**> sizes;

transform(words.begin(), words.end(), back\_inserter(sizes), len); // 5, 6, 5

// ... використання вектору sizes

**return** 0;

}

Існує альтернативний варіант алгоритму transform() для роботи з двома вихідними послідовностями. Цей алгоритм буде розглянуто нижче.

**Сортування послідовностей**

За допомогою алгоритму sort() можна здійснити сортування масивів і контейнерів з довільним доступом. Застосовується алгоритм швидкого сортування. Наприклад, так можна організувати сортування вектору цілих чисел за зростанням значень:

vector<**int**> v;

v.push\_back(3);

v.push\_back(4);

v.push\_back(1);

sort(v.begin(), v.end()); // 1, 3, 4

За допомогою функції sort() можна також сортувати масиви. Як і для інших алгоритмів, замість ітераторів ми використовуємо вказівник (ім'я масиву):

**const** **int** n = 3;

**int** a[n] = { 3, 4, 1 };

sort(a, a + n); // 1, 3, 4

У випадку використання більш складних типів, для яких не визначені відносини "більше" і "менше", функцію sort() потрібно викликати з трьома параметрами. Третій параметр – функція-предикат – функція, що повертає результат типу **bool**. Відповідна функція для алгоритму sort() повинна приймати два параметри типу елементів послідовності та повертати **true** (1), якщо елементи не потрібно змінювати місцями, або **false** (0) в іншому випадку. Можна також змінити критерій сортування. В такий спосіб можна здійснити сортування вектору цілих чисел за зменшенням:

#include <algorithm>

#include <vector>

**using** std::vector;

**using** std::sort;

**bool** decrease(**int** m, **int** n)

{

**return** m > n;

}

**int** main()

{

vector<**int**> v = { 1, 11, 7, 4, 8 };

sort(v.begin(), v.end(), decrease);

// ... Використання відсортованого вектору

**return** 0;

}

*Примітка*: функція sort() не дозволяє сортувати списки (std::list), оскільки відповідний клас не надає необхідних ітераторів.

**Функціональні об'єкти. Функції-предикати. Адаптери функцій**

Критерії, на підставі яких працює більшість алгоритмів, обчислюються за допомогою застосування операції виклику функції до об'єкту, переданому в цей алгоритм як параметр. Таким параметром може бути вказівник на функцію або функціональний об'єкт. Основний недолік безпосереднього використання вказівника на функцію під час роботи з алгоритмами полягає в неможливості збільшення кількості параметрів функції.

***Функціональний об'єкт*** (***функтор***, functor) є екземпляром деякого класу, в якому за допомогою функції-елементу перевантажена операція "круглі дужки" (виклик функції). Звернення до цієї операції здійснюється кожен раз, коли функціональний об'єкт використовується як функція. Клас, що описує тип функціонального об'єкта, може мати елементи даних і конструктор, в якому ці елементи даних ініціалізуються необхідними значеннями. Ці значення використовуються в функції, перевантажують операцію "круглі дужки". У наведеному нижче прикладі функціональний об'єкт зберігає посилання на потік виведення:

#include <iostream>

#include <fstream>

#include <algorithm>

#include <vector>

#include <functional>

**using** std::vector;

**using** std::ostream;

**using** std::ofstream;

**using** std::for\_each;

**using** std::endl;

**class** WriteInteger

{

ostream& out;

**public**:

WriteInteger(ostream& strm) : out(strm) { }

**void** **operator** () (**const** **int**& i) { out << i << endl; }

};

**void** main()

{

vector<**int**> a({ 1, 2, 3, 4 }); // ініціалізація списком значень доступна у версії C++11

// Виведення вектора:

ofstream out("result.txt");

for\_each(a.begin(), a.end(), WriteInteger(out));

}

***Предикат*** – це функція (або функціональний об'єкт), що повертає значення типу **bool**. Низку предикатів, відповідних операцій відносини і логічних операцій, реалізовано в Стандартній бібліотеці. Для їх використання треба підключати заголовний файл <functional>. До цих предикатів відносяться унарний предикат logical\_not (який реалізує !) і бінарні предикати:

* equal\_to (==)
* not\_equal\_to (!=)
* greater (>)
* less (<)
* greater\_equal (>=)
* less\_equal (<=)
* logical\_and (&&)
* logical\_or (||).

Кожен такий предикат є шаблоном, які приймає як параметр тип значення, для якого здійснюється операція. У наведеному нижче прикладі використовується предикат logical\_not() в алгоритмі transform():

vector<**int**> a = { 0, 1, 0, 2, 0, 3 };

vector<**int**> b(6); // результуюча послідовність

transform(a.begin(), a.end(), b.begin(), logical\_not<**int**>());

// b: 1 0 1 0 1 0

В заголовному файлі <functional> Стандартної бібліотеки визначені арифметичні функції, доступні як функціональні об'єкти. Можна використовувати такі операції:

* plus (додавання, x + y)
* minus (віднімання, x - y)
* multiplies (множення, x \* y)
* divides (ділення, x / y)
* modulus (залишок від ділення, x % y)
* negate (унарний мінус, -x).

За допомогою арифметичних функціональних об'єктів можна виконувати зазначені дії над всіма елементами однієї або двох послідовностей. Наприклад, можна використовувати інший варіант алгоритму transform(). Цей варіант алгоритму приймає дві вхідних послідовності, визначені трьома ітераторами, і результуючу (визначену одним ітератором):

**int** main()

{

**int** arr[] = { 0, 1, 2, 3, 4, 5, 5, 4, 3, 2, 1, 0 };

vector<**int**> a(arr, arr + 6);

vector<**int**> b(arr + 6, arr + 12);

vector<**int**> c;

transform(a.begin(), a.end(), b.begin(), back\_inserter(c), plus<**int**>());

// Клас WriteInteger було реалізовано раніше:

for\_each(c.begin(), c.end(), WriteInteger(cout));

**return** 0;

}

Функція back\_inserter() додає елементи в кінець контейнера, збільшуючи його до необхідних розмірів.

***Адаптери функцій*** доступні через заголовний файл <functional>.

***Зв'язувач*** (binder) дозволяє використовувати функціональний об'єкт з двома аргументами як функцію з одним аргументом шляхом зв'язування одного аргументу зі значенням. Функції bind1st() і bind2nd() отримують як аргументи бінарну функцію (функціональний об'єкт) f і значення x і повертають відповідно класи binder1st і binder2nd. Функціональний об'єкт повинен бути класом, побудованим з класу binary\_function. Клас binder1st прив'язує значення до першого аргументу бінарної функції, а binder2nd робить те ж саме для другого аргументу.

Наприклад, для знаходження першого елемента вектору цілих чисел, більшого 10, можна скористатися функцією

find\_if(vi.begin(), vi.end(), bind2nd(less<**int**>(), 10));

***Адаптер функцій-елементів*** дозволяє використовувати вказівник на функцію-елемент як аргумент алгоритму. Для цього використовуються шаблонні функції mem\_fun, і mem\_fun\_ref. Перша функція застосовується для виклику функцій-елементів через вказівник на об'єкт. Друга функція використовується для звернення до функції через посилання на об'єкт.

Наприклад, якщо в векторі зберігаються вказівники на геометричні фігури (клас Shape, що має функцію-елемент draw()), то за допомогою

for\_each(vs.begin(), vs.end(), mem\_fun(&Shape::draw));

можна зобразити всі фігури.

***Адаптер вказівника на функцію*** дозволяє використовувати вказівник на функцію як аргумент алгоритму. Це необхідно для реалізації зв'язувачів, оскільки вони повинні зберігати копію аргументу для подальшого використання. Шаблонна функція ptr\_fun() використовує класи pointer\_to\_unary\_function і pointer\_to\_binary\_function для інкапсуляції вказівників на функції з відповідною кількістю параметрів.

***Заперечувач*** дозволяє висловити протилежний предикат. Функціональні адаптери not1 і not2 використовують для інвертування унарного й бінарного функціонального об'єктів відповідно. В наведеному нижче прикладі для алгоритму sort() визначено ознаку "не менше" як умову того, що послідовність відсортована. Така послідовність буде відсортована за зменшенням:

sort(a.begin(), a.end(), not2(less<**int**>()));

**Приклади програм**

**Використання векторів**

У наведеній нижче програмі створено вектор векторів цілих, значення яких виводяться на екран:

#include <iostream>

#include <vector>

**using** std::cout;

**using** std::vector;

**void** main()

{

vector<vector<**int**> > a = { vector<**int**>({ 1, 2 }), vector<**int**>({ 3, 4 }) };

**for** (**auto** &row : a)

{

**for** (**auto** &x : row)

{

cout << x << "\t";

}

cout << "\n";

}

}

**Робота з чергою з пріоритетом**

Припустимо, є клас для представлення країни:

#include <iostream>

#include <queue>

#include <string>

**using** std::string;

**class** Country

{

**private**:

string name;

**double** square;

**long** population;

**public**:

Country() : name(""), square(1), population(0) { }

Country(string n, **double** s, **long** p) : name(n), square(s), population(p) { }

string getName() **const** { **return** name; }

**double** getSquare() **const** { **return** square; }

**long** getPopulation() **const** { **return** population; }

**void** setName(string value) { name = value; };

**void** setSquare(**double** value) { square = value; }

**void** setPopulation(**long** value) { population = value; }

**double** density() **const** { **return** population / square; }

};

Реалізуємо чергу з пріоритетом, в яку помістимо об'єкти типу Country. Більш пріоритетною при вилученні з черги вважається країна з більшою щільністю населення. Для того, щоб помістити об'єкти типу Country в таку чергу, необхідно для них визначити операцію <. Ця операція може бути реалізована зовнішньої функцією. Така функція навіть може не бути другом класу:

**bool** **operator**<(**const** Country& c1, **const** Country& c2)

{

**return** c1.density() < c2.density();

}

Тепер об'єкти можна помістити в чергу з пріоритетом:

**using** std::priority\_queue;

**using** std::cout;

**using** std::endl;

**int** main()

{

Country c0("Ukraine", 603700, 42539000);

Country c1("France", 544000, 57804000);

Country c2("Sweden", 450000, 8745000);

Country c3("Germany", 357000, 81338000);

priority\_queue<Country> cq;

cq.push(c0);

cq.push(c1);

cq.push(c2);

cq.push(c3);

**while** (cq.size())

{

cout << cq.top().getName() << endl;

cq.pop();

}

**return** 0;

}

Альтернативою використанню перевантаженої операції < є використання функціонального об'єкта – об'єкта класу, що має перевантажену операцію (). Ім'я цього класу необхідно вказати як третій параметр конструктору priority\_queue:

**class** Less

{

**public**:

**bool** **operator**()(**const** Country& c1, **const** Country& c2) **const**

{

**return** c1.density() < c2.density();

}

};

**int** main()

{

. . .

priority\_queue<Country, std::vector<Country>, Less> cq;

. . .

}

**Робота з асоціативним масивом**

Припустимо, необхідно прочитати з файлу "values.txt" цілі значення, порахувати кількість кожного із значень і вивести на екран значення і кількість їх повторень.

#include <iostream>

#include <fstream>

#include <map>

**using** std::map;

**using** std::ifstream;

**using** std::cout;

**using** std::endl;

**int** main()

{

map<**int**, **int**> m;

**int** i;

{

ifstream in("values.txt");

**while** (in >> i)

m[i]++;

}

map<**int**, **int**>::iterator mi;

**for** (mi = m.begin(); mi != m.end(); mi++)

cout << mi->first << " " << mi->second << endl;

**return** 0;

}

Як розвиток цього прикладу можна запропонувати використання функціонального об'єкта, який задає протилежну логіку сортування елементів.

*Для самостійного вивчення*: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

*Література*

1. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL:  <http://www.ph4s.ru/bookprogramir_1.html>
3. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>
4. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL:  <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>
5. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997.  URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>.
6. *Герб Саттер,Андрей Александреску* "Стандарты программирования на С++ ", [*Вильямс, 2005*](http://www.williamspublishing.com/)*; 304 с.*
7. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*
8. *Скотт Мейерс.* Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.*"ДМК",2000;304 с.*
9. Г.Буч. Об‘єктно-орієнтоване проектування з прикладами застосування. – К.: Видавничий центр "Академія". 2002. ­ 499 с.
10. Страуструп Б. Язык программирования С++· Киев: "ДиаСофт", 1993. - 256 с.

*Контрольні запитання для самоперевірки*.

1. Які основні будівельні блоки STL ви можете визначити?
2. Що таке контейнер Стандартної бібліотеки?
3. Як у Стандартній бібліотеці представлені рядки символів?
4. Чим список відрізняється від масиву?
5. Що таке адаптер послідовності?
6. Чим черга відрізняється від стеку?
7. Що таке асоціативний масив?
8. Де застосовують асоціативні масиви?
9. Чим множина відрізняється від інших контейнерів?
10. Чим відрізняються послідовні й асоціативні контейнери?
11. Що таке ітератор?
12. Що таке алгоритм Стандартної бібліотеки?