Тема 2. Використання стандартної бібліотеки шаблонів STL

1. Огляд стандартної бібліотеки шаблонів

Перелічимо основні можливості стандартної бібліотеки C++, які з'явилися в ній з розвитком самої мови C++: потоки, числові методи (наприклад, операції з комплексними числами, множення масиву на константу та інші нескладні методи), рядки (класи для роботи з текстовою інформацією), множини, контейнери, алгоритми й об'єкти-функції, ітератори та інше.

Стандартна бібліотека шаблонів STL [14-21, 30] – частина стандартної бібліотеки C++ SCL, що забезпечує загальноцільові стандартні класи й функції, які реалізують найбільш популярні й широко використовувані алгоритми й структури даних.

Бібліотека STL розроблена співробітником Hewllet-Packard Олександром Степановим. STL будується на основі шаблонів класів, і тому вхідні в неї алгоритми й структури застосовні майже до всіх типів даних. Ядро бібліотеки утворюють три складові: контейнери, алгоритми й ітератори.

Контейнери (containers) – це об'єкти, призначені для зберігання інших елементів. Тут представлені різні варіанти часто використовуваних структур даних – вектори, списки, стеки, черги й т.п. Точніше, навіть не самі структури, а заготівки для них. Конкретну структуру для свого типу даних програміст утворює сам за допомогою стандартних контейнерів.

Ітератори (iterators) – це об'єкти, які стосовно контейнера відіграють роль вказівників. Вони дозволяють одержати доступ до вмісту контейнера приблизно так само, як вказівники використовуються для доступу до елементів масиву. Ітератори застосуються як зв'язок між структурами даних (не обов'язково контейнерами) і алгоритмами або іншим кодом, який ці структури використовує.

Алгоритми (algorithms) виконують операції над містимим контейнера. Існують алгоритми для ініціалізації, сортування, пошуку, заміни вмісту контейнерів. Багато алгоритмів призначені для роботи з послідовністю (sequence), що являє собою лінійний список елементів усередині контейнера. Велика частина алгоритмів STL побудована по єдиному принципу. Алгоритм отримує на вхід пару ітераторів (інтервал) і для елементів з цього інтервалу виконує деяку задачу, наприклад, сортування.

Шаблони контейнерів, алгоритми та й взагалі вся бібліотека винесені в окремий простір імен, який одержав назву std. Існує кілька способів користуватися ключовими словами із простору імен std:

- після всіх файлів, що підключаються (наприклад, #include<set>), використати директиву

using namespace std;

- підключити бажані модулі STL окремими директивами using:

using std::stack;

using std::find;

- вказувати оператор прив'язки std:: перед кожним STL типом даних або алгоритмом.

2. Контейнери

Контейнер – це сховище елементів і засобів, щоб із цими елементами працювати. Всі контейнери розроблені таким чином, щоб при необхідності можна було працювати з елементами незалежно від того, у якому саме різновиді контейнера – векторі, списку, чи черзі – ці елементи зберігаються. Природно, різновиди розрізняються між собою й внутрішнім устроєм, і ефективністю різних операцій, і набором додаткових, характерних тільки для них способів доступу до елементів. Наприклад, у вектора є оператор індексування [], якого немає у списків, проте список дозволяє ефективно виконувати операції вставки й видалення елементів. Але при цьому кожній з контейнерів підтримує набір однотипних операцій – невеликий, але достатній для того, щоб на його основі можна було писати узагальнені алгоритми для роботи з елементами.

У мові С був лише один контейнер – масив. У мові С++ контейнери реалізовані у вигляді шаблонів класів, при цьому тип елемента задається параметром шаблона.

Стандартна бібліотека шаблонів містить наступні контейнерні класи: vector (динамічний масив), string (рядок), list (список), deque (дек), set (множина), multiset (мультимножина), bitset (множина бітів), map (асоціативний масив), multimap (мульти-асоціативний масив). Крім того, класи queue (черга), priority_queue (черга з пріоритетом) і stack (стек) не є окремими контейнерами, а побудовані на базі інших контейнерів. Для використання контейнерів необхідно підключити однойменний заголовний файл (наприклад, для використання списків варто використовувати #include <list>).

Розглянемо далі основні контейнери, структури і класи стандартної бібліотеки.

vector<T> – контейнерний шаблон, що описує динамічний одновимірний масив елементів типу T. Визначений у файлі заголовків <vector>. Відмінна риса від інших типів контейнерів – наявність ефективної операції доступу по індексу (operator[]). Цей контейнер не є зручним при вставці й видаленні елементів.

Наведемо приклад роботи з вектором і матрицею з використанням бібліотеки STL.

```
#include <vector>

#include <iostream>

// використовується простір імен бібліотеки STL

using namespace std;

. . .

// створюється вектор з 10 елементів типу int

vector<int> v(10);

for (int i=0; i<10; i++)

v[i] = i;  // використовується operator[]

// створюється матриця 10´10 елементів типу int

vector<int> m[10];

for(int i=0;i < 10;i++)

for(int j=0;j < 10;j++)

// використовується член-фунція push_back

m[i].push_back(j);

for(int i=0;i < 10;i++)

{

for(int j=0;j < 10;j++) cout<<m[i][j]<<" ";

cout<<endl;

}

. . .
```

Матрицю можна описати і так:

```
vector< vector<int> > m;
```

Оскільки дві кутові дужки, що йдуть підряд без пробілу, більшість трансляторів сприймають як операцію «<<» або «>>», то, при наявності в коді вкладених STL конструкцій, між кутовими дужками треба ставити пробіл.

Для спрощення описів можна визначити типи, наприклад так:

typedef vector<int> tvector; // тип вектор

typedef vector<tvector> tmatrix; // тип матриця

string. Для роботи з рядками передбачений шаблонний клас basic_string з якого визначається клас string:

typedef basic_string <char> string;

Клас string відрізняється від vector<char> функціями для маніпулювання рядками й політикою роботи з пам'яттю. Для визначення довжини рядка, використовується string::length(), а не vector::size().

Розглянемо приклад роботи з рядками.

string sl = "слово";

string s1 = sl.substr(0, 3),   // "сло"

s2 = sl.substr(1, 3),   // "лов"

s3 = sl.substr(0, s.length()-1),   //"слов"

s4 = sl.substr(1),   // "лово"

s = "Завтра - термін здачі лабораторної !";

// пошук першого пробілу

int ind1 = s.find_first_of(' ');

// пошук першого пробілу, знаків ! або -

int ind2 = s.find_first_of(" -!");

Також рядки можна додавати за допомогою operator+, порівнювати за допомогою операторів порівняння. Таким чином, vector<string> можна впорядковувати стандартними методами. Рядки порівнюються в лексикографічному порядку.

pair<T1,T2> – пара. Шаблонна структура, що містить два поля, можливо, різних типів. Поля мають назви first і second. Прототип пари виглядає так:

template<class T1, class T2> struct pair {

T1 first;

T2 second;

pair() {}

pair(const T1& x, const T2& y) :

first(x), second(y) {}

};

Прикладом контейнера пара є опис pair<int,int> – два цілих числа. Приклад складеної пари: pair<string, pair<int,int> > – рядок і два цілих числа. Використовувати подібну пару можна, наприклад, так:

pair<string, pair<int,int> > P;

string s = P.first; // Рядок

int x = P.second.first; // Перше ціле

int y = P.second.second; // Друге ціле

Об'єкти pair можна порівнювати по полях у порядку опису пар ліворуч праворуч. Тому пари активно використовуються як усередині бібліотеки STL, так і програмістами у своїх цілях.

Наприклад, якщо необхідно впорядкувати точки на площині по полярному куту, то можна помістити всі точки в структуру виду

vector< pair<double, pair<int,int> >,

де double – полярний кут точки, а pair<int,int> – її координати. Після цього викликати стандартну функцію сортування, і точки будуть упорядковані по полярному куту.

Пари активно використовуються в асоціативних масивах.

list<T> – контейнерний шаблон, що описує список елементів типу T. Визначається у файлі заголовків <list>. Зручний при частих вставках і видаленнях елементів.

deque<T> – дек (черга із двома кінцями і елементами типу T). Робота з елементами на обох кінцях (але не в середині) майже так само ефективна, як у списку, а доступ по індексу [] – як у векторі. Слабке місце – вставка й видалення елемента всередині.

stack<T>, **queue<T**> – стек і черга з елементів типу T. Визначаються у файлах заголовків відповідно <stack> і <queue>. Контейнер, на базі якого будуються стек і черга, є параметром шаблона. За замовчуванням стек і черга використовують для реалізації дек (тоді в кутових дужках задається лише один параметр – тип елементів T, другий параметр – за замовченням deque<T>). Однак, це можна поміняти, указавши другим (необов'язковим) параметром тип іншого контейнера. Для стека це може бути список або вектор (обидва підтримують push_back), а для черги – список (підтримує pop_front):

stack<int,vector<int> > s;

queue<double,list<double> > q;

set<T>– контейнерний шаблон, що описує множину елементів типу T. Визначаються у файлі заголовку <set> .

map <T1,T2> – контейнерний шаблон, що описує асоціативний масив. Асоціативний масив – це масив, у якого як індекс типу T1 (ключ) може використовуватися значення нецілочислового типу, наприклад, рядок. T2 – це тип самих елементів. Асоціативний масив багато в чому схожий на звичайну послідовність, але дозволяє працювати з парами джерело-значення. Втім, у деяких асоціативних контейнерах значень, як таких, немає – є тільки ключі.

3. Конструктори та члени-функції контейнерів

Для створення будь-якого з контейнерів може бути використаний конструктор за замовчуванням (у цьому випадку створюється порожній контейнер), конструктор копіювання, конструктор, який ініціалізує контейнер діапазоном елементів іншого контейнера (не обов'язково того ж типу):

container<T> c;

container<T> c2(c);

container<T> c1(b,e);

Тут container – будь-який з типів контейнерів STL, а c, c1 і c2 – утворені контейнери.

Загальні функції для всіх контейнерів

c.begin()	повертає ітератор початку контейнера
c.end()	повертає ітератор кінця контейнера
c.rbegin()	повертає зворотний ітератор початку контейнера
c.rend()	повертає зворотний ітератор кінця контейнера
c.size()	повертає розмір контейнера
c.empty()	перевіряє, чи порожній контейнер
c.clear()	очищає контейнер, роблячи його розмір рівним 0
c=c2	присвоює вміст контейнера c2 однотипному контей-неру c. Використов
c.assign(b,e)	заміняє вміст контейнера елементами діапазону [b,e) іншого контейне
c.swap(c1)	міняє місцями вміст двох однотипних контейнерів (відбувається обмін
c==c1 c!=c1	порівнює контейнери на рівність і нерівність, використовуючи операції
c<c1 c>c1 c<=c1 c>=c1	порівнює контейнери лексикографічно

Функції для контейнерів vector і list

container<T> c(n)	конструктор контейнера з n елементів зі значенням за замовчуван
container<T> c(n,t)	конструктор контейнер з n елементів, заповнених значенням t
c.insert(it,t)	вставляє елемент t у контейнер c перед ітератором it. Повертає іт
c.insert(it,n,t)	вставляє n елементів t у контейнер c перед ітератором it. Поверта
c.insert(it,b,e)	вставляє діапазон елементів [b,e) у контейнер c перед ітератором елемента.

c.erase(it)	видаляє елемент у контейнері c в позиції ітератора it. Повертають
c.erase(b,e)	видаляє діапазон елементів [b,e) у контейнері c Повертають ітера
c.assign(n,t)	присвоює контейнеру c n копій t
c.front()	повертає посилання на перший елемент
c.back()	повертає посилання на останній елемент
c.push_back(t)	додає t у кінець контейнера c
c.pop_back()	видаляє останній елемент. Повертає void

Функції для контейнерів vector, string і deque

c[i]	здійснює довільний доступ до i-того елемента без перевірки вихо
c.at(i)	здійснює довільний доступ з перевіркою виходу за межі (у випадк

Функції для контейнерів vector і string

c.resize(n)	змінює розмір
c.capacity()	повертає пам'ять, відведену під масив
c.reserve(n)	резервує пам'ять в n елементів

Функції для контейнерів list

c.sort()	сортування списку c
c.sort(Cmp)	сортування списку c із критерієм сортування Cmp (аналог опер
c.push_front(t)	додавання елемента t у початок
c.pop_front()	видалення першого елемента
c.splice(it,c1)	переміщення всіх елементів зі списку c1 у список c перед ітера
c.splice(it,c1,it1)	переміщення елемента *it1 зі списку c1 у список c перед ітерат
c.splice(it,c1,b,e)	переміщення елементів [b,e) зі списку c1 у список c перед it
c.merge(c2)	об'єднання двох сортованих списків з видаленням елементів з
c.remove(t)	видалення всіх елементів t
c.remove_if(Pred)	видалення всіх елементів, що задовольняють предикату Pred
c.unique()	видалення рядом розташованих дублікатів, використовуючи =
c.unique(BinPred)	видалення рядом розташованих дублікатів, використовуючи бі порівняння на рівність

4. Ітератори

Ітератори – це щось, аналогічне вказівнику на елемент масиву. По суті, вказівник на елемент масиву – теж ітератор. Маючи ітератор p якого-небудь контейнера, можна перейти до наступного або попереднього елемента (++p, --ip), одержати сам елемент (*p), порівняти з іншим ітератором того ж контейнера (p==p1). Але ітератори відрізняються від звичайних вказівників звуженими можливостями. Наприклад, не з всіма типами ітераторів можна робити операцію декремента. Деякі ітератори дозволяють працювати з об'єктами в режимі "тільки читання" або "тільки запис".

Отже, ітератори потрібні, щоб давати зручний і однаковий доступ до елементів будь-якого контейнера. Їхнє головне призначення – задавати послідовності елементів, тобто – ті границі (по елементах), у межах яких треба працювати. Для звичайного масиву можна обійтися не ітераторами, а індексами. Але, якщо треба вибрати з асоціативного масиву із ключем-рядком всі елементи з індексом, що починається на A, то треба використати ітератор.

Існує багато різновидів ітераторів – прямі, зворотні, двунапрямні, з довільним (випадковим) доступом та інші [19].

Тип ітератора	***Доступ***	***Розіменування***	***Ітерація***	***Порівняння***
Ітератор виводу (output iterator)	Тільки запис	*	++	
Ітератор введення (input iterator)	Тільки читання	*, ->	++	==, !=
Прямий ітератор (Forward iterator)	Читання й запис	*, ->	++	==, !=
Двунапрямний ітератор (bidirectional iterator)	Читання й запис	*, ->	++, --	==, !=
Ітератор з довільним доступом (random-access iterator)	Читання й запис	*, ->, []	++, ---,+, -, +=, -=	==, !=, <, <=, >, >=

У різних алгоритмах використовуються різні типи ітераторів.

Ітератор зовсім необов'язково використовувати з повною версією стандартного контейнера. Деякі алгоритми працюють і зі звичайними масивами, при цьому в якості ітераторів їм передаються вказівники на елементи масиву.

Ітератори контейнерів реалізовані у вигляді шаблонів, причому вони є частиною визначення класу самого контейнера. Описати ітератор можна, наприклад, так:

list<int>::iterator p;

vector<int>::revers_iterator r;

Тепер можна звертатися до елементів списка через ітератор

list<int> res = v;

for (p=res.begin(); p!=res.end(); p++)

cout << *p << endl;

Тут v – будь-який контейнер, наприклад, вектор.

5. Алгоритми

Алгоритми (і об'єкти-функції) потрібні для того, щоб можна було ефективно працювати з набором елементів контейнера. Але, не тільки контейнера. Завдяки ітераторам алгоритми є досить незалежними, їх можна відокремити від поняття "контейнер". Алгоритми просто обробляють елементи послідовності, заданої ітераторами.

Що саме треба проробити з послідовністю, задається самим алгоритмом. Поведінкою багатьох алгоритмів можна керувати, передаючи їм як аргументи функції (це можуть бути звичайні функції або функції-об'єкти).

З використанням алгоритмів можливе створення дуже могутніх і ефективних програм. По компактності такий код перевершує код, написаний на таких сучасних мовах, як Java і C#, і в значній мірі ефективніше останнього.

STL-алгоритми визначені в заголовному файлі <algorithm>. Вони представляють набір готових функцій, які можуть бути застосовані до STL-контейнерів (і не тільки) і можуть бути розділені на групи. Перелічимо основні алгоритми в групах.

Алгоритми, що не модифікують послідовностей елементів

for_earch ()	виконує операції для кожного елемента послідовності
find ()	знаходить перше входження значення в послідовність
find_if ()	знаходить першу відповідність предикату в послідовності
count ()	підраховує кількість входжень значення в послідовність
count_if ()	підраховує кількість виконань предиката в послідовності
search ()	знаходить перше входження послідовності як підпослідовності
search_n ()	знаходить n-те входження значення в послідовність

Алгоритми, що модифікують послідовностей елементів

copy()	копіює послідовність, починаючи з першого елемента
swap()	міняє місцями два елементи
replace()	заміняє елементи із зазначеним значенням
replace_if()	заміняє елементи при виконанні предиката
replace_copy()	копіює послідовність, замінюючи елементи з зазначеним значенням
replace_copy_if()	копіює послідовність, замінюючи елементи при виконанні предиката
fill()	заміняє всі елементи даним значенням
remove()	видаляє елементи з даним значенням
remove_if()	видаляє елементи при виконанні предиката
remove_copy()	копіює послідовність, видаляючи елементи з зазначеним значенням
remove_copy_if()	копіює послідовність, видаляючи елементи при виконанні предиката
reverse()	міняє порядок проходження елементів на зворотний
random_shuffle()	переміщає елементи відповідно до випадкового

	рівномірному розподілу (“тасує” послідовність)
transform()	виконує задану операцію над кожним елементом послідовності
unique()	видаляє рівні сусідні елементи
unique_copy()	копіює послідовність, видаляючи рівні сусідні елементи

Алгоритми сортування членів послідовностей

sort ()	сортує послідовність із задовільною середньої ефективністю
partial_sort ()	сортує частину послідовності
stable_sort ()	сортує послідовність, зберігаючи порядок слідування рівних елементів
lower_bound ()	знаходить перше входження значення у відсортованій послідовності
upper_bound ()	знаходить перший елемент, більший ніж задане значення
binary_search ()	визначає, є чи даний елемент у відсортованій послідовності
merge ()	зливає дві відсортовані послідовності

Алгоритми роботи з множинами

includes ()	перевірка на входження
set_union ()	об'єднання множин
set_intersection ()	перетин множин
set_difference ()	різниця множин

Мінімуми й максимуми

min ()	менше із двох
max ()	більше із двох
min_element ()	найменше значення в послідовності
max_element ()	найбільше значення в послідовності

Перестановки

next_permutation ()	наступна перестановка в лексикографічному порядку
pred_permutation ()	попередня перестановка в лексикографічному порядку

Для того, щоб використовувати ці та інші алгоритми треба мати відповідну документацію. У середовище C++Builder 6 включено документацію по бібліотеці STL. Необхідну інформацію можна також знайти на сайтах [16-21].

Для багатьох алгоритмів STL необхідно задати умову, за допомогою якої алгоритм визначає, що йому необхідно робити з тим або іншим членом контейнера. За означенням, предикат – це

функція, що приймає один або більше параметрів і повертає значення істина або неправда. Існує набір стандартних предикатів.

6. Приклад використання контейнерів, ітераторів, алгоритмів і предикатів

Продемонструємо використання стандартної бібліотеки шаблонів STL на приклалі контейнера vector. Для цього створимо декілька об'єктів різними типами конструкторів, застосуємо до них алгоритми for_each, find, find_if (з користувацьким предикатом), find_end, reverse і функцію swap.

```
#include <iostream>

#include <vector> // для контейнера vector

// для алгоритмів for_each, find, find_if, find_end

#include <algorithm>

// використовується простір імен бібліотеки STL

using namespace std;

// функції, які застосовуються в алгоритмах

void mul2(int& i) { i *= 2; }

void print(int i) { cout << i << endl; }

// предикат для алгоритму find_if

bool negative(int x) { return x<0; }

int main(int argc, char* argv[]) {

// створюється вектор з 10 елементів

vector<int> v(10);

// використовується operator[]

for (int i=0; i<10; i++) v[i] = i;

v[6]=-4;

// створюється контейнер - копію першого

// (конструктор копіювання)

vector<int> res = v;

// використовується ітератор для множення
```

```
// всіх елементів res на 2 і друку масиву
vector<int>::iterator p;
for (p=res.begin(); p!=res.end(); p++)
{
*p *= 2;
cout << *p << endl;
}
// аналогічні дії робимо за допомогою
// стандартного алгоритму for_each
for_each(res.begin(), res.end(), mul2);
for_each(res.begin(), res.end(), print);
// стандартний алгоритм  find (шукає у масиві число 3)
p = find(v.begin(), v.end(), 3);
if (p != v.end())cout << "\nЗнайдено " << *p << endl;
// пошук першого від'ємного елемента
// (використовується предикат negative)
p= find_if(v.begin(), v.end(), negative);
if (p != v.end())
cout << " Перший від\'ємний  елемент " << *p << endl;
else  cout << " Від\'ємних елементів не знайдено \n" ;
int subv[]={24,28,32,36};
// ще один тип конструктора
vector<int> sub(subv,subv+4);
// пошук в res останньої послідовності sub
p=find_end(res.begin(),res.end(),sub.begin(),sub.end());
if(p!= res.end())cout << "Послідовність знайдено \n";
else  cout << " Послідовність не знайдено \n" ;
res.swap (sub); //перестановка об\'єктів res і sub
```

```
reverse (res);  // перестановка елементів вектора res
cout << " Перетворений вектор res: " << endl;
for_each(res.begin(), res.end(), print);
//затримка екрану з результатами
char c;          cin>>c;
return 0;
}
```