УДК 004
П.А. Пех, О.В. Кузава, М.О.Ковальський,
Луцький національний технічний університет

ОСОБЛИВОСТІ ПРОГРАМУВАННЯ ЗАСОБАМИ МОВИ С++ З ВИКОРИСТАННЯМ БІБЛІОТЕКИ ШАБЛОНІВ STL

У роботі проаналізовані особливості програмування засобами мови С++ з використанням бібліотеки шаблонів STL.
Ключові слова: програмування мовою С++, бібліотека шаблонів STL.
Табл. 2. Літ. 2.

Постановка проблеми. Стандартна бібліотека шаблонів (***Standard Template Library***, ***STL*****)** — бібліотека шаблонів мови С++, що містить набір узгоджених узагальнених алгоритмів, контейнерів, засобів доступу до їхнього вмісту і різних допоміжних функцій [1,2]. Шаблони дозволяють визначати функції, що параметризуються, і класи, у яких параметрами служать не змінні, а типи. З огляду на це, заслуговує особливої уваги проблема застосування стандартної бібліотеки шаблонів у процесі програмування засобами мови С++, як така, що недостатньо досліджена вітчизняними та зарубіжними вченими.

Аналіз останніх досліджень та публікацій. Питанням програмування мовою С++ присвячено багато праць [1,2], однак у цих роботах не приділялося достатньо уваги використанню стандартної бібліотеки шаблонів.

Невирішені частини проблеми. В даній роботі запропоновано один із можливих шляхів інтенсифікації процесу програмування мовою С++ - за рахунок використання бібліотеки шаблонів ***STL***. На наш погляд, вирішення цієї проблеми має певне теоретичне і прикладне значення.

Метою дослідження є технології використання бібліотеки шаблонів ***STL*** з метою інтенсифікації процесу програмування мовою С++.

Основні результати дослідження. У бібліотеці шаблонів виділяють п'ять основних компонентів:

1. Контейнер ***(container)*** - зберігання набору об'єктів в пам'яті.
2. Ітератор ***(iterator)*** - забезпечення засобів послідовного доступу до вмісту контейнера.
3. Алгоритм ***(algorithm)*** - визначення обчислювальної процедури.
4. Адаптер ***(adaptor)*** - адаптація компонентів для забезпечення різного інтерфейсу.
5. Функціональний об'єкт ***(functor)*** - приховування функції в об'єкті для використання іншими компонентами.

Детальніше зупинимося на перелічених вище базових поняттях бібліотеки шаблонів, оскільки саме вони і складають ядро бібліотеки ***STL.*** Класи бібліотеки ***STL*** знаходяться у відповідних заголовкових файлах. Зокрема, клас `vector` для роботи з контейнерами, знаходиться у заголовковому файлі `vector`. Для роботи з алгоритмами у програму слід включити заголовковий файл `<algorithm>`. У разі використання вбудованих функторів, у програму включається заголовок `<functional>`. Наведемо приклади оголошення контейнерів типу **vector.**

```
vector <char> v1;  //порожній контейнер-вектор;
vector <char> v2('z');  //контейнер-вектор, заповнений символом z;
vector <char> v3=v2; //контейнер-вектор, що складається з елементів
іншого
                     //контейнера-вектора;
char mas[]="strichka";
vector <char> v4(mas,mas+3); //контейнер-вектор, або ж стрічка "str"
```

Контейнери бібліотеки ***STL*** (табл. 1) займають у ній центральне місце. Їх можна розділити на чотири категорії: послідовні, асоціативні, контейнери-адаптери і псевдоконтейнери.

Таблиця 1. **Контейнери бібліотеки шаблонів STL**

Контейнер	Опис
Послідовні контейнери	
vector	С-подібний динамічний масив довільного доступу з автоматичною зміною розміру в процесі додавання/вилучення елементів. Елементи масиву займають неперервну частину оперативної пам'яті. Операція додавання/вилучення елемента в кінець масиву потребує часу ***O(1).*** Операція додавання/вилучення елемента на початок або в середину масиву потребує часу ***O(n).*** Існує спеціальний шаблон **vector** для типу **bool**, який вимагає менше пам'яті за рахунок зберігання даних типу **bool** у вигляді бітів.
list	Двозв'язний список, елементи якого на відміну від контейнера **vector** можуть зберігатися у довільних, не обов'язково неперервних, частинах оперативної пам'яті. Для данного типу контейнерів характерним є швидка вставка та вилучення елемента списку за час ***O(1),*** повільний пошук елементів і доступ до них за час ***O(n).***
deque	Подібний до контейнера **vector,** але забезпечує можливість швидкої вставки і вилучення елементів, як на початок, так і в кінець масиву
Асоціативні контейнери	
set	Впорядкована множина унікальних елементів. Під час вставки/ вилучення елементів множини ітератори, що вказують на елементи цієї множини, не стають недійсними. Контейнер дозволяє виконання стандартних операцій, що є характерними для множин – перетин множин, об'єднання множин, віднімання множин. Тип елементів множини повинен реалізовувати оператора порівняння `operator<,` або ж потрібно надати функцію-компаратор. Контейнер реалізований на основі самобалансуючого дерева двійкового пошуку.
multiset	Контейнер, подібний до **set**, але дозволяє зберігати елементи, що повторюються
map	Впорядкований асоціативний масив пар елементів, що складаються з ключів і відповідних їм пар значень. Ключі мають бути унікальними. Порядок проходження елементів визначається ключами. При цьому тип ключа повинен реалізовувати оператора порівняння `operator<,` або ж потрібно надати функцію-компаратор.
multimap	Подібний на контейнер **map,** але дозволяє зберігати ключі,
Контейнери-адаптери	
stack	Стек – це контейнер, у якому додавання та вилучення елементів здійснюється лише з одного кінця
queue	Черга – це контейнер, у якому з одного кінця можна додавати елементи, а з іншого - вилучати
priority_queue	Черга з пріоритетом – це контейнер, організований так, що найбільший елемент завжди стоїть на першому місці
Псевдоконтейнери	
bitset	Контейнер, що служить для зберігання бітових масок. Подібний до **vector <bool>** фіксованого розміру. Розмір фіксується тоді, коли оголошується об'єкт **bitset.** Ітераторів у цього контейнера немає. Оптимізований за розміром пам'яті.
basic_string	Контейнер, що служить для зберігання та оброблення рядків. Зберігає в пам'яті елементи підряд єдиним блоком, що дозволяє швидкий доступ до всієї послідовності.

Нижче наведена програма, яка демонструє роботу з одновимірним цілочисельним масивом за допомогою контейнера `vector.` Програма передбачає використання контейнера-вектора `vec,` запис у нього значень 1..5, виведення на друк розміру масиву і виведення значень його елементів.

```
#include <iostream>
#include <vector>
usung namespase std;
void main()  {
  vector <int> vec; // оголошено контейнер-вектор (масив)
  for (int i=1;i<6; i++)  vec.push_back(i);// додати в масив числа
1..5
  cout<<vec.size()<<endl; // виведення розміру масиву
```

```
  for (i=0;i<vec.size(); i++)  cout<<vec[i]<<" "; // друк елементів
масиву
}
```

У бібліотеці ***STL*** для доступу до елементів як посередник використовується узагальнена абстракція, що іменується ітератором. Ітератор – це об'єкт, який забезпечує доступ до елементів контейнера. Його можна розглядати як узагальнене поняття «вказівник на елемент». Якщо елемент має тип `T`, то тип ітератора буде `T*`. Отже, кожен контейнер підтримує «свій» вид ітератора, який є «модернізованим» інтелектуальним вказівником, що «знає» як отримати доступ до елементів цього конкретного контейнера. Наприклад, друк елементів цілочисельного одновимірного масиву можна виконати за допомогою ітератора:

```
for (vector <char>::iterator i=vec.begin(); i !=;i<vec.end(); i++)
  cout<< *i;
```

Над контейнерами можна виконувати операції, наведені в табл. 2. Як показано у цій таблиці, вектори, двосторонні черги і списки підтримують різний набір операцій.

Таблиця 2. **Операції над векторами, двосторонніми чергами та списками**

Операція	Функція	vector	deque	list
Додавання в кінець	push_back	+	+	+
Видалення з кінця	pop_back	+	+	+
Додавання в початок	push_front	-	+	+
Видалення з початку	pop_front	-	+	+
Додавання в будь-яке місце	insert	(+)	(+)	+
Видалення з будь-якого місця	erase	(+)	(+)	+
Відсортувати	sort	+	+	-

Плюс в дужках (+) означає, що функції **insert** і **erase**, хоча і визначені для векторів і двосторонніх черг, але час виконання для цих контейнерів набагато більший, ніж для списків. Кажуть, що їх виконання займає лінійний час для векторів і двосторонніх черг, а це означає: час їх виконання пропорційний довжині послідовності, що зберігається в контейнері. На противагу цьому, всі операції, позначенні знаком + (без дужок), виконуються за постійний час, тобто час, необхідний для їх виконання, не залежить від довжини послідовності.

Засоби **STL** дозволяють зменшити кількість компонентів. Наприклад, замість написання окремої функції пошуку елементу для кожного типу контейнера забезпечується єдина версія, яка працює з кожним з них, поки дотримуються основні вимоги. Далі розглядуються алгоритми для виконання типових задач з залученням засобів ***STL.***

Алгоритм **`sort`** для сортування елементів вектора**.** Цей алгоритм зчитує `n` елементів і добавляє їх в контейнер. Застосовуючи вираз `sort(a.begin(0, a.end()),` виконуємо сортування елементів. Використовується два значення ітераторів в якості аргументів: `a.begin(),` який посилається на перший елемент вектора, та `a.end(),` який посилається на наступний після останнього елемента вектора.

```
vector <int> a;
int n,x;
cin>>n;
for (int i=0; i<n; i++) {
 cin>>x;
 a.push_back(x);
}
sort(a.begin(), a.end());
for (int i=0; i<n; i++) {
 cout<<a[i]<<" ";
}
cout<<endl;
```

Алгоритм **`find`** для знаходження потрібного значення вектора. Потрібне значення у векторі можна знайти за допомогою фрагмента

```
vector <int>::iterator pos=find(a.begin, a.end(),x);
```

Алгоритм **copy** для копіювання елементів одного контейнера в інший. У даному випадку джерелом може бути вектор, а приймачем – список, як це показано нижче.

```
int a[4] ={10,20,30,40};
vector <int> v(a, a+4);
list <int> L(4);;
copy(v.begin(),v.end(), L.begin()); // Результат L: 10 20 30 40
```

Алгоритм **merge** забезпечує об' єднання двох контейнерів в один. Фрагмент, наведений нижче, об'єднує контейнери a: 2 3 8 20 25 та b: 7 9 23 28 30 33 і утворює результуючий контейнер c: 2 3 7 8 9 20 23 25 28 30 33.

```
int numbers[]={2,3,8,20,25,7,9,23,28,30,33};
vector <int> a(numbers,numbers+5);  // a: 2 3 8 20 25
vector <int> b(numbers+5,numbers+11);  // b: 7 9 23 28 30 33
vector <int> c(a.size()+b.size());
merge(a.begin(), a.end(), b.begin(), b.end(), c,begin());
// Результат c: 2 3 7 8 9 20 23 25 28 30 33
```

Алгоритм **replace** дозволяє знайти суму всіх елементів послідовності в заданому контейнері і замінити їх іншим значенням.

```
char str[]="abcabcabc";
int n=strlen(str);
replace(str,str+n, 'b','q');
cout<<str<<endl; //Результат aqcaqcaqc
```

Алгоритм **reverse** дозволяє легко замінити задану послідовність на обернену їй. Це реалізовано у наступному фрагменті.

```
char str[]="abcabcabc";
int n=strlen(str);
reverse(str,str+n);
cout<<str<<endl; //Результат cbacbacba
```

Алгоритм **accumulate** дозволяє обчислити суму всіх елементів послідовності або ж підпослідовності. Третій аргумент задає початкове значення для процесу сумування.

```
const int N=5;
int a[N]={4,12,3,6,10}, sum=0;
sum=accumulate(a,a+N,0); // Результат: 35
sum accumulate(a,a+N,1000); // Результат: 1035
```

Алгоритм **count** дозволяє обчислити кількість елементів, що дорівнюють певному значенню.

```
char str[]="This demonsrates the Standard Template Library";
int n=strlen(str);
int number=count(str,str+n, 'e'); // Результат: 5
```

Насамкінець, розглянемо два шляхи розв'язання однієї і тієї ж задачі – традиційним шляхом і за допомогою бібліотеки **STL:** дано послідовність із n натуральних чисел. Необхідно підрахувати у ній кількість унікальних значень. Нижче наведено варіант розв'язку задачі без використання ***STL.***

```
/*************************************************************/
/* Рзв'язання задачі без використання бібліотеки  шаблонів STL */
/*************************************************************/
#include <iostream.h>
#include <conio.h>
int main() {
```

```
 clrscr();
 int n, res=0, f;
 cout<<"  Введіть кількість членів послідовності: ";
 cin>>n;
 int *a=new int[n];

 for (int i=0; i<n; i++) {
  cout<<"  Введіть значеня a["<<i<<"]: ";
  cin>>a[i];
 }

 for ( i=0; i<n; i++) {
  f=1;
  for (int j=0; j<i; j++)
   if (a[i] == a[j]) f=0;
  if (f) res++;
 }

 cout<<"  Кількість унікальних значень в послідовності res="<<res<<endl;
 getch();
 return 0;
}
```

```
Результати роботи програми:
  Введіть кількість членів послідовності: 5
  Введіть значеня a[0]: 1
  Введіть значеня a[1]: 1
  Введіть значеня a[2]: 2
  Введіть значеня a[3]: 2
  Введіть значеня a[4]: 3
  Кількість унікальних значень в послідовності res=3
```

Далі наведено варіант розв'язку задачі з використанням ***STL.***

```
/*************************************************************/
/* Рзв'язання задачі з використанням бібліотеки шаблонів STL */
/*************************************************************/
#include <iostream.h>
#include <algorithm.h>
#include <map>
#include <conio.h>
using namespace std;
int main() {
 clrscr();
 int n, res, x;
 map<int, bool> a;
 cout<<"  Введіть кількість членів послідовності: ";
 cin>>n;

 for (int i=0; i<n; i++) {
  cout<<"  Введіть значеня x: ";
  cin>>x;
  a[x]=1;
 }
 res=a.size();
 cout<<"  Кількість унікальних значень в послідовності res="<<res<<endl;
 getch();
 return 0;
}
```

Висновок. Використання бібліотеки шаблонів значно прискорює процес програмування мовою С++.

1. Бондарев В.М. Программирование на С++. -Харьков: ТОВ «Компанія СМИТ», 2004.- 567 с
2. Шилдт Г. Полный справочник по С++. -4-е издание. –М: Издательский дом «Вильямс», 2004. – 786 с.