Міністерство освіти і науки України
Національний університет “Львівська політехніка”
Інститут комп’ютерних наук та інформаційних технологій
Кафедра програмного забезпечення

Звіт
Про виконання лабораторних робіт №11

На тему:

«Стандартна бібліотека шаблонів. Контейнери та алгоритми.»

Лектор:
доцент каф. ПЗ
Коротєєва Т.О.
Виконав:
ст. гр. ПЗ-11
Морозов О. Р.
Прийняла:
доцент каф. ПЗ
Коротєєва Т.О.

« __ » _____ 2022 р.

∑ = _____ _____ .

Львів – 2022

Тема: Стандартна бібліотека шаблонів. Контейнери та алгоритми..
Мета: Навчитись використовувати контейнери стандартної бібліотеки шаблонів та вбудовані алгоритми.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Стандартна бібліотека шаблонів (скор. “STL” від “Standard Template Library”) — це частина Стандартної бібліотеки C++, яка містить набір шаблонів контейнерних класів (наприклад, std::vector і std::array), алгоритмів і ітераторів.

Безумовно, найбільш використовуваним функціоналом бібліотеки STL є контейнерні класи (або як їх ще називають — «контейнери»). Бібліотека STL містить багато різних контейнерних класів, які можна використовувати в різних ситуаціях. Якщо говорити в загальному, то контейнери STL діляться на три основні категорії:

- послідовні;
- асоціативні;
- адаптери.

Послідовні контейнери (або «контейнери послідовності») — це контейнерні класи, елементи яких знаходяться в послідовності. Їх визначальною характеристикою є те, що ви можете додати свій елемент в будь-яке місце контейнера. Найбільш поширеним прикладом послідовного контейнера є масив: при додаванні 4 елементів в масив, ці елементи перебуватимуть (в масиві) в точно такому ж порядку, в якому ви їх додали.

Починаючи з C++11, STL містить 6 контейнерів послідовності:

- std::vector;
- std::deque;
- std::array;
- std::list;
- std::forward_list;
- std::basic_string.

Асоціативні контейнери — це контейнерні класи, які автоматично сортують всі свої елементи (в тому числі і ті, які додаєте ви). За замовчуванням асоціативні контейнери виконують сортування елементів, використовуючи оператор порівняння <.

- set — це контейнер, в якому зберігаються тільки унікальні елементи, і повторення заборонені. Елементи упорядковано відповідно до їх значень.
- multiset — це set, але в якому допускаються повторювані елементи.

- map (або «асоціативний масив») — це set, в якому кожен елемент є парою “ключ-значення”. “Ключ” використовується для сортування та індексації даних і повинен бути унікальним, а “значення” — це фактичні дані.
- multimap (або «словник») — це map, який допускає дублювання ключів. Всі ключі відсортовані в порядку зростання, і ви можете подивитися значення по ключу.

Адаптери — це спеціальні визначені контейнерні класи, які адаптовані для виконання конкретних завдань. Найцікавіше полягає в тому, що ви самі можете вибрати, який послідовний контейнер повинен використовувати адаптер.

- stack (стек) — це контейнерний клас, елементи якого працюють за принципом LIFO (англ. «Last In, First Out» = «останнім прийшов, першим пішов»), тобто елементи додаються (вносяться) в кінець контейнера і видаляються (виштовхуються) звідти ж (з кінця контейнера). Зазвичай в стеках використовується deque в якості послідовного контейнера за замовчуванням (що трохи дивно, оскільки vector був би більш підходящим варіантом), але ви також можете використовувати vector або list.
- queue (черга) — це контейнерний клас, елементи якого працюють за принципом FIFO (англ. «First In, First Out» = «першим прийшов, першим пішов»), тобто елементи додаються (вносяться) в кінець контейнера, але видаляються (виштовхуються) з початку контейнера. За замовчуванням в черзі використовується deque в якості послідовного контейнера, але також може використовуватися і list.
- priority_queue (черга з пріоритетом) — це тип черги, в якій всі елементи відсортовані (за допомогою оператора порівняння <). При додаванні елемента, він автоматично сортується. Елемент з найвищим пріоритетом (найбільший елемент) знаходиться на самому початку черги з пріоритетом, також, як і видалення елементів виконується з самого початку черги з пріоритетом.

Ітератор — це об’єкт, здатний перебирати елементи контейнерного класу без необхідності користувачеві знати реалізацію цього контейнерного класу. У багатьох контейнерах (особливо в списку і в асоціативних контейнерах) ітератори є основним способом доступу до елементів цих контейнерів.

Крім контейнерів і ітераторів, бібліотека STL також надає ряд універсальних алгоритмів для роботи з елементами контейнерів. Вони дозволяють виконувати такі операції, як пошук, сортування, вставка, зміна позиції, видалення і копіювання елементів контейнера.

Алгоритми STL реалізовані у вигляді глобальних функцій, які працюють з використанням ітераторів. Це означає, що кожен алгоритм потрібно

реалізувати всього лише один раз, і він працюватиме з усіма контейнерами, які надають набір ітераторів (включаючи і ваші власні (користувацькі) контейнерні класи). Хоча це має величезний потенціал і надає можливість швидко писати складний код, у алгоритмів також є і “темна сторона” — деяка комбінація алгоритмів і типів контейнерів може не працювати/працювати з поганою продуктивністю/викликати нескінченні цикли, тому слід бути обережним.

Бібліотека STL надає досить багато алгоритмів. На цьому уроці ми розглянемо лише деякі з найбільш поширених і простих у використанні алгоритмів. Для їх роботи потрібно підключити заголовок algorithm.

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

Написати програму з використанням бібліотеки STL.

В програмі реалізувати наступні функції:

1. Створити об’єкт-контейнер (1) у відповідності до індивідуального варіанту і заповнити його даними користувацького типу, згідно варіанту.

2. Вивести контейнер.

3. Змінити контейнер, видаливши з нього одні елементи і замінивши інші.

4. Проглянути контейнер, використовуючи для доступу до його елементів ітератори.

5. Створити другий контейнер цього ж класу і заповнити його даними того ж типу, що і перший контейнер.

6. Змінити перший контейнер, видаливши з нього n елементів після заданого і добавивши опісля в нього всі елементи із другого контейнера.

7. Вивести перший і другий контейнери.

8. Відсортувати контейнер по спаданню елементів та вивести результати.

9. Використовуючи необхідний алгоритм, знайти в контейнері елемент, який задовільняє заданій умові.

10. Перемістити елементи, що задовільняють умові в інший, попередньо пустий контейнер (2). Тип цього контейнера визначається згідно варіанту.

11. Проглянути другий контейнер.

13. Відсортувати перший і другий контейнери по зростанню елементів, вивести результати.

15. Отримати третій контейнер шляхом злиття перших двох.

16. Вивести на екран третій контейнер.

17. Підрахувати, скільки елементів, що задовільнюють заданій умові, містить третій контейнер.

Оформити звіт до лабораторної роботи. Звіт має містити варіант завдання, код розробленої програми, результати роботи програми (скріншоти), висновок.

Завдання згідно варіанту

Варіант 2

2	stack	queue	float

Код програми

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
#include <QLayout>

#include <stack>
#include <queue>
#include <iterator>
#include <algorithm>
#include <vector>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE
```

```
class Widget : public QWidget
{
 Q_OBJECT

public:
 Widget(QWidget *parent = nullptr);

 std::stack<float, std::vector<float>> stack_1;
 std::stack<float, std::vector<float>> stack_2;
 std::queue<float> queue;
 std::vector<float> vector;

private slots:
 void OnStart();

private:
 QLabel *taskResult_1;
 QLabel *outputLabel_1;
 QLabel *taskResult_2;
 QLabel *outputLabel_2;
 QLabel *taskResult_3;
 QLabel *outputLabel_3;
 QLabel *taskResult_4;
 QLabel *outputLabel_4;
 QLabel *taskResult_5;
 QLabel *outputLabel_5;
 QLabel *taskResult_6;
 QLabel *outputLabel_6;

 QPushButton * startButton;
};
#endif // WIDGET_H
```

widget.cpp

```
#include "widget.h"
#include "ui_widget.h"

#define N 7

template <typename Type1, typename Type2>
QString showStack(std::stack<Type1, Type2> &stk) {
QString stack;
```

```
std::stack<Type1, Type2> outstk = stk;
for(size_t i = 0; i < stk.size(); i++){
	stack += QString::number(outstk.top()) += "  ";
	outstk.pop();
}
return stack;
}

template <typename Type1>
QString showQueue(std::queue<Type1> &queue) {
QString q;
std::queue<Type1> outqueue = queue;
for(size_t i = 0; i < queue.size(); i++){
	q += QString::number(outqueue.front()) += "  ";
	outqueue.pop();
}
return q;
}

template <class T, class U, class Compare>
void sort_stack(std::stack<T,U> &stack, Compare comp)
{
std::vector<T> tmp_container;
tmp_container.reserve(stack.size());
while(!stack.empty())
{
	tmp_container.push_back(std::move(stack.top()));
	stack.pop();
}
std::sort(tmp_container.begin(), tmp_container.end(), comp);
for(auto it:tmp_container)
{
	stack.push(std::move(it));
}
}
template <class T, class U>
void sort_stack(std::stack<T,U> &stack)
{
sort_stack(stack, std::less<T>());
}

template <class T, class U, class Compare>
void sort_stack2(std::stack<T,U> &stack, Compare comp)
{
std::vector<T> tmp_container;
tmp_container.reserve(stack.size());
while(!stack.empty())
{
	tmp_container.push_back(std::move(stack.top()));
	stack.pop();
}
```

```
	std::sort(tmp_container.begin(), tmp_container.end(), comp);
	for(auto it:tmp_container)
	{
		stack.push(std::move(it));
	}
	}
	template <class T, class U>
	void sort_stack2(std::stack<T,U> &stack)
	{
	sort_stack(stack, std::greater<T>());
	}

	void Widget::OnStart() {
	bool ch = true;
	ch = stack_1.empty();
	qDebug() << ch;
	ch = stack_2.empty();
	qDebug() << ch;
	ch = queue.empty();
	qDebug() << ch;
	vector.clear();

	for(int i = 0; i < N; i++) {
		stack_1.push(i + 0.2*i);
		if(!i%2){
		stack_2.push(stack_1.top()+0.3);
		} else {
		stack_2.push(stack_1.top()-0.4);
		}
	}


this->outputLabel_1->setText((showStack(stack_1))+"\n"+(showStack(stack_2)));
	//-------------------------------------------------------------------------
-------------------//
	std::stack<float, std::vector<float>> stack_1_copy;
	for(int i = 0; i < N; i++) {

		if((abs(i)-2) && (abs(i)-3)){
			stack_1_copy.push(stack_1.top());
		}
		stack_1.pop();
	}

	stack_1 = stack_1_copy;

	this->outputLabel_2->setText(showStack(stack_1));
	//-------------------------------------------------------------------------
-------------------//

	while(!stack_1_copy.empty()){
```

```
            stack_1_copy.pop();
        }

        qDebug() << stack_1_copy.empty();

        std::stack<float, std::vector<float>> stack_2_copy = stack_2;
        for(size_t i = 0; i < stack_2.size(); i++) {
            stack_1_copy.push(stack_2_copy.top());
            stack_2_copy.pop();
        }

        for(size_t i = 0; i < stack_2.size(); i++) {
            stack_1.push(stack_1_copy.top());
            stack_1_copy.pop();
        }
        stack_1_copy = stack_1;
        sort_stack(stack_1);
        this->outputLabel_3->setText(showStack(stack_1));
        //------------------------------------------------------------------------
-------------------//

        std::stack<float, std::vector<float>> stack_1_copy2 = stack_1_copy;
        for(size_t i = 0; i < stack_1_copy.size(); i++) {
            stack_1.push(stack_1_copy.top());
            stack_1_copy.pop();
        }

        for(size_t i = 0; i < stack_1.size(); i++) {
            queue.push(stack_1.top());
            stack_1.pop();
        }

        this->outputLabel_4->setText(showQueue(queue));
        //------------------------------------------------------------------------
-------------------//

        stack_1 = stack_1_copy2;
        sort_stack2(stack_1);
        this->outputLabel_5->setText(showStack(stack_1));

        //------------------------------------------------------------------------
-------------------//

        while(!queue.empty()){
            vector.push_back(queue.front());
            queue.pop();
        }

        while(!stack_1.empty()){
            vector.push_back(stack_1.top());
            stack_1.pop();
```

```
        }

        int i = 0;
        for(std::vector<float>::iterator it = vector.begin(); it != vector.end();
++it, ++i){
                if(i == 9) {this->outputLabel_6->setText(this->outputLabel_6->text()
+ ("\n"));}
                this->outputLabel_6->setText(this->outputLabel_6->text() +
QString::number(*it) + "   ");
        }


        qDebug() << "Stack 1" << showStack(stack_1) << "\nStack 1 copy" <<
showStack(stack_1_copy) << "\nStack 1 copy2" << showStack(stack_1_copy2);

        }

        Widget::Widget(QWidget *parent)
        : QWidget(parent)
        {
        QGridLayout * layout = new QGridLayout;

        this->setFixedSize(600,600);
        this->setWindowIcon(QIcon("://img//Stack.png"));
        this->setWindowTitle("STL containers");
        this->setStyleSheet
                        (
                                "QPushButton{ background-color: #fff; border: 1px solid
#dbdbdb; border-radius: .375em; color: #363636;"
                                "font-family: Segoe UI; font-size: 1rem; height:
2.5em; line-height: 1.5; padding: calc(0.5em - 1px) 1em;"
                                "position: relative; text-align: center;
vertical-align: top; white-space: nowrap; }"

                                "QPushButton::hover{ border-color: #5500ff; outline:
0;}"

                                "QPushButton::pressed { border-color: #4a4a4a; outline:
0;}"
                                );

        this->startButton = new QPushButton(QIcon("://img//Caret down.png"),
"Start");

        this->taskResult_1 = new QLabel("Stack 1\nStack 2");
        this->outputLabel_1 = new QLabel;

        this->taskResult_2 = new QLabel("Stack 1\nchanged");
        this->outputLabel_2 = new QLabel;

        this->taskResult_3 = new QLabel("Stack 1+2\nsorted to lower");
```

```
this->outputLabel_3 = new QLabel;

this->taskResult_4 = new QLabel("Queue");
this->outputLabel_4 = new QLabel;

this->taskResult_5 = new QLabel("Stack\nsorted to large");
this->outputLabel_5 = new QLabel;

this->taskResult_6 = new QLabel("Stack + Queue\nIn vector");
this->outputLabel_6 = new QLabel;

layout->addWidget(this->taskResult_1, 0, 0);
layout->addWidget(this->taskResult_2, 1, 0);
layout->addWidget(this->taskResult_3, 2, 0);
layout->addWidget(this->taskResult_4, 3, 0);
layout->addWidget(this->taskResult_5, 4, 0);
layout->addWidget(this->taskResult_6, 5, 0);

layout->addWidget(this->outputLabel_1, 0, 1);
layout->addWidget(this->outputLabel_2, 1, 1);
layout->addWidget(this->outputLabel_3, 2, 1);
layout->addWidget(this->outputLabel_4, 3, 1);
layout->addWidget(this->outputLabel_5, 4, 1);
layout->addWidget(this->outputLabel_6, 5, 1);

layout->addWidget(this->startButton, 0, 2);

connect(this->startButton, &QPushButton::clicked, this, &Widget::OnStart);

setLayout(layout);
}
```

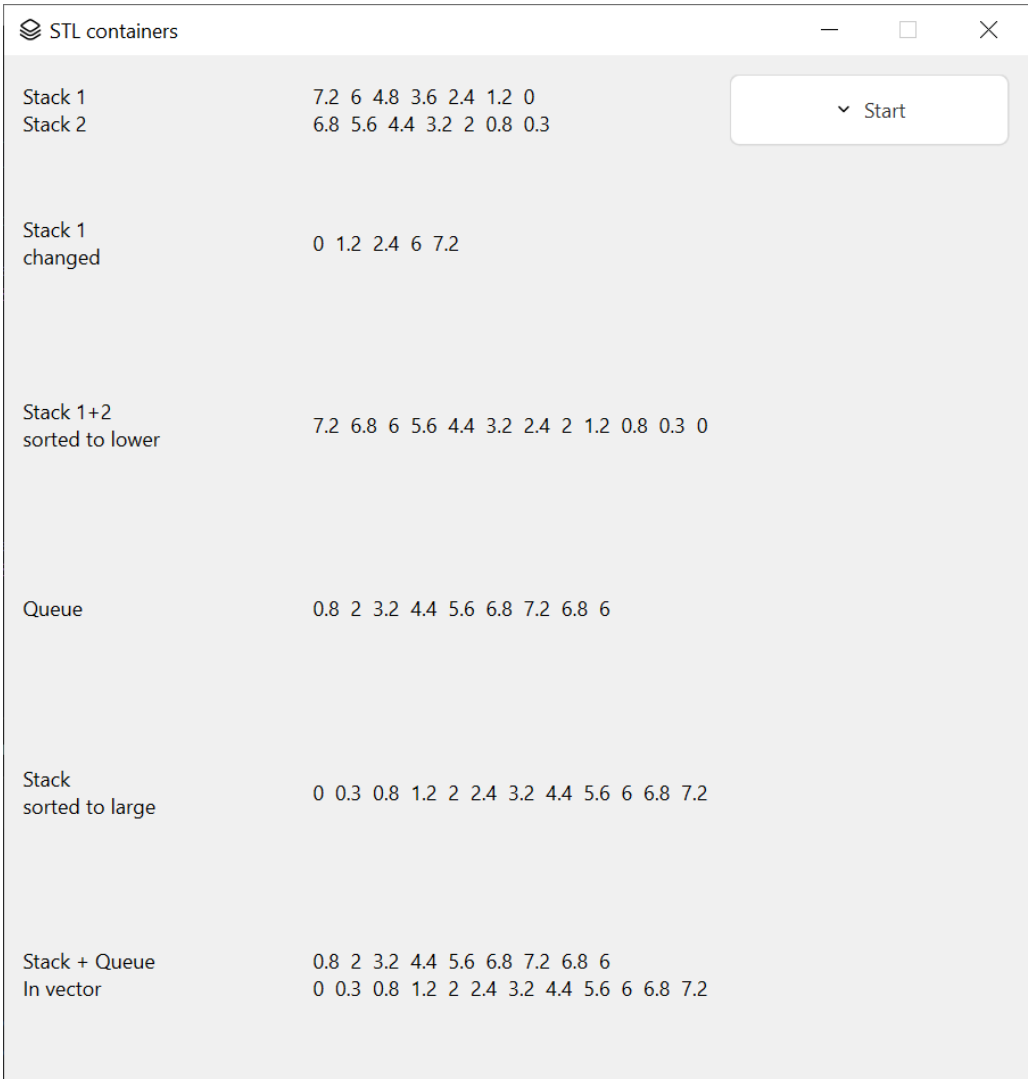
main.cpp

```
#include "widget.h"

#include <QApplication>

int main(int argc, char *argv[]) {
 QApplication a(argc, argv);
 Widget w;
 w.show();;
 return a.exec();
}
```

Вигляд програми



ВИСНОВКИ

Виконуючи лабораторну роботу №11, я навчився використовувати контейнери стандартної бібліотеки шаблонів та вбудовані алгоритми