**Міністерство освіти і науки України**

**Національний університет “Львівська політехніка”**

**Інститут комп’ютерних наук та інформаційних технологій**

***Кафедра ПЗ***

**Звіт**

до лабораторної роботи №10

на тему «Шаблони класів»

з дисципліни “Об’єктно-орієнтоване програмування”

**Виконав:**

студент групи ПЗ-11

Солтисюк Д.А

**Перевірив**:

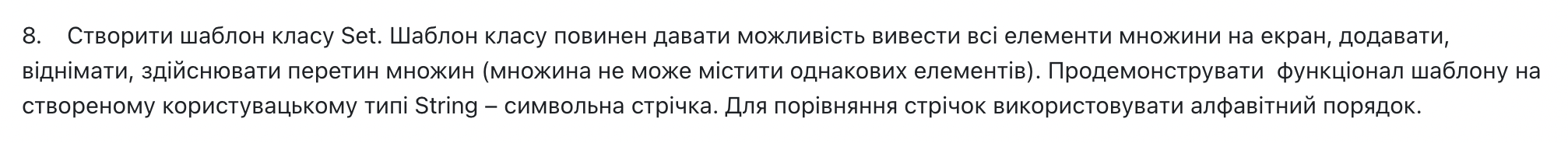
 доц. Коротєєва Т.О.

Львів

**2022**

##### **Тема.** Шаблони класів. **Мета.** Навчитись створювати шаблони класу та екземпляри шаблонів.

##### **Завдання для лабораторної роботи:**

Створити шаблон класу та продемонструвати його роботу за індивідуальним варіантом. Оформити звіт до лабораторної роботи. Звіт має містити варіант завдання, код розробленої програми, результати роботи програми (скріншоти), висновок.

**Теоретичні відомості:**

Прикладом шаблону в реальному житті є трафарет — об’єкт, в якому прорізаний малюнок/візерунок/символ. Якщо прикласти трафарет до іншого об’єкту і розпорошити фарбу, то отримаємо цей же малюнок фарбою, якщо засипати піском, то отримаємо інший варіант того самого малюнка і т.п. Ми зможемо зробити десятки таких малюнків різних кольорів і різних основ! При цьому нам потрібен лише один трафарет.

У мові C++ шаблони функцій — це функції, які служать взірцем для створення інших подібних функцій. Головна ідея — створення функцій без вказівки точного типу(ів) деяких або всіх змінних. Для цього ми визначаємо функцію, вказуючи тип параметра шаблону, який використовується замість будь-якого типу даних. Після того, як ми створили функцію з типом параметра шаблону, ми фактично створили «трафарет функції».

При виклику шаблону функції, компілятор використовує «трафарет» в якості зразка функції, замінюючи тип параметра шаблону на фактичний тип змінних, переданих у функцію!

Розробка шаблонів функцій дозволяє створювати узагальнені за алгоритмом функції, які можуть працювати для різних типів даних (як для вбудованих, та і для користувацьких).

Для оголошення шаблону функції використовується ключове слово template , далі в трикутних дужках записується тип параметру шаблону <typename T> або <class T> . У мові C++ прийнято називати типи параметрів шаблонів великою літерою T, але можна використовувати будь-який ідентифікатор.

Якщо потрібно кілька типів параметрів шаблону, то вони розділяються комами: template <typename T1, typename T2>

Коли компілятор зустрічає виклик шаблону функції, він копіює шаблон функції і замінює типи параметрів шаблону функції фактичними (переданими) типами даних. Функція з фактичними типами даних називається екземпляром шаблону функції (або *«об’єктом шаблону функції»*).

Якщо створити шаблон функції, але не викликати його, то екземпляри цього шаблону створені не будуть.

Шаблони функцій працюють як з вбудованими типами даних (char, int, double тощо), так і з класами. Екземпляр шаблону компілюється як звичайна функція. Будь-які оператори або виклики інших функцій, які присутні в шаблоні функції, повинні бути визначені для роботи з фактичними типами даних.

Переваги: Шаблони функцій економлять багато часу, тому що шаблон ми пишемо тільки один раз, а використовувати можемо з різними типами даних. Шаблони функцій набагато спрощують подальшу підтримку коду, і вони безпечніші, тому що немає необхідності виконувати вручну перевантаження функції, копіюючи код і змінюючи лише типи даних, коли потрібна підтримка нового типу даних.

У шаблонів функцій є кілька недоліків:

По-перше, деякі старі компілятори можуть не підтримувати шаблони функцій або підтримувати, але з обмеженнями. Однак зараз це вже не така проблема, як раніше.

По-друге, шаблони функцій часто видають божевільні повідомлення про помилки, які набагато складніше розшифрувати, ніж помилки звичайних функцій.

По-третє, шаблони функцій можуть збільшити час компіляції і розмір коду, тому що один шаблон може бути «реалізований» і перекомпільований в декількох файлах.

Дані недоліки досить незначні в порівнянні з потужністю і гнучкістю шаблонів функцій!

**Результат:**

custom-set.h

#pragma once

#include <algorithm>

#include <functional>

#include <iostream>

#include <iterator>

#include <vector>

**using** std::vector;

template <typename T> class CustomSet {

**using** ElementsType = vector<T>;

private:

ElementsType elements;

public:

*// iterator interface*

typename ElementsType::iterator begin() { **return** elements.begin(); }

typename ElementsType::iterator end() { **return** elements.end(); }

CustomSet(ElementsType initial) { **this**->elements = initial; };

ElementsType getElements() { **return this**->elements; }

void push(T element) {

**if** (std::find(std::begin(**this**->elements), std::end(**this**->elements),

element) != std::end(**this**->elements)) {

**return**; *// found duplicate*

}

**this**->elements.push\_back(element);

}

void erase() { **this**->elements.clear(); }

void erase(T element) {

**this**->elements.erase(std::remove(std::begin(**this**->elements),

std::end(**this**->elements), element),

std::end(**this**->element));

}

ElementsType intersection(CustomSet<T> \*otherSet) {

ElementsType v1 = **this**->elements;

ElementsType v2 = otherSet->getElements();

ElementsType v3; *// Intersection of V1 and V2*

std::sort(v1.begin(), v1.end());

std::sort(v2.begin(), v2.end());

std::set\_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(),

back\_inserter(v3));

**return** v3;

}

};

custom-string.cpp

#include "custom-string.h"

*// Overloading the assignment operator*

CustomString &CustomString::operator=(const CustomString &rhs) {

**if** (**this** == &rhs)

**return** \***this**;

**delete**[] str;

str = **new** char[strlen(rhs.str) + 1];

strcpy(str, rhs.str);

**return** \***this**;

}

*// Overloading the plus operator*

CustomString operator+(const CustomString &lhs, const CustomString &rhs) {

int length = strlen(lhs.str) + strlen(rhs.str);

char \*buff = **new** char[length + 1];

*// Copy the strings to buff[]*

strcpy(buff, lhs.str);

strcat(buff, rhs.str);

buff[length] = '\0';

*// String temp*

CustomString temp{buff};

*// delete the buff[]*

**delete**[] buff;

*// Return the concatenated string*

**return** temp;

}

*// Overloading the stream*

*// extraction operator*

istream &operator>>(istream &is, CustomString &obj) {

char \*buff = **new** char[1000];

memset(&buff[0], 0, **sizeof**(buff));

is >> buff;

obj = CustomString{buff};

**delete**[] buff;

**return** is;

}

*// Overloading the stream*

*// insertion operator*

ostream &operator<<(ostream &os, const CustomString &obj) {

os << obj.str;

**return** os;

}

*// Function for swapping string*

void CustomString::swp(CustomString &rhs) {

CustomString temp{rhs};

rhs = \***this**;

\***this** = temp;

}

*// Function to copy the string*

void CustomString::copy(char s[], int len, int pos) {

**for** (int i = 0; i < len; i++) {

s[i] = str[pos + i];

}

s[len] = '\0';

}

*// Function to implement push\_bk*

void CustomString::push\_bk(char a) {

*// Find length of string*

int length = strlen(str);

char \*buff = **new** char[length + 2];

*// Copy character from str*

*// to buff[]*

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

buff[i] = str[i];

}

buff[length] = a;

buff[length + 1] = '\0';

*// Assign the new string with*

*// char a to string str*

\***this** = CustomString{buff};

*// Delete the temp buff[]*

**delete**[] buff;

}

*// Function to implement pop\_bk*

void CustomString::pop\_bk() {

int length = strlen(str);

char \*buff = **new** char[length];

*// Copy character from str*

*// to buff[]*

**for** (int i = 0; i < length - 1; i++)

buff[i] = str[i];

buff[length - 1] = '\0';

*// Assign the new string with*

*// char a to string str*

\***this** = CustomString{buff};

*// delete the buff[]*

**delete**[] buff;

}

*// Function to implement get\_length*

int CustomString::get\_length() { **return** strlen(str); }

*// Function to illustrate Constructor*

*// with no arguments*

CustomString::CustomString() : str{nullptr} {

str = **new** char[1];

str[0] = '\0';

}

*// Function to illustrate Constructor*

*// with one arguments*

CustomString::CustomString(char \*val) {

**if** (val == nullptr) {

str = **new** char[1];

str[0] = '\0';

}

**else** {

str = **new** char[strlen(val) + 1];

*// Copy character of val[]*

*// using strcpy*

strcpy(str, val);

str[strlen(val)] = '\0';

}

}

*// Function to illustrate*

*// Copy Constructor*

CustomString::CustomString(const CustomString &source) {

str = **new** char[strlen(source.str) + 1];

strcpy(str, source.str);

}

*// Function to illustrate*

*// Move Constructor*

CustomString::CustomString(CustomString &&source) {

str = source.str;

source.str = nullptr;

}

custom-string.h

#pragma once

#include <cstring>

#include <iostream>

**using namespace** std;

class CustomString {

*// Prototype for stream insertion*

friend ostream &operator<<(ostream &os, const CustomString &obj);

*// Prototype for stream extraction*

friend istream &operator>>(istream &is, CustomString &obj);

*// Prototype for '+'*

*// operator overloading*

friend CustomString operator+(const CustomString &lhs,

const CustomString &rhs);

char \*str;

public:

*// No arguments constructor*

CustomString();

*// pop\_back() function*

void pop\_bk();

*// push\_back() function*

void push\_bk(char a);

*// To get the length*

int get\_length();

*// Function to copy the string*

*// of length len from position pos*

void copy(char s[], int len, int pos);

*// Swap strings function*

void swp(CustomString &rhs);

*// Constructor with 1 arguments*

CustomString(char \*val);

*// Copy Constructor*

CustomString(const CustomString &source);

*// Move Constructor*

CustomString(CustomString &&source);

*// Overloading the assignment*

*// operator*

CustomString &operator=(const CustomString &rhs);

*// Destructor*

~CustomString() { **delete** str; }

};

main.cpp

#include <QApplication>

#include "widget.h"

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

QApplication a(argc, argv);

Widget w;

w.show();

**return** a.exec();

}

widget.cpp

#include "widget.h"

#include "custom-set.h"

#include "custom-string.h"

#include <iostream>

#include <string>

#include <vector>

**using** std::string;

void Widget::onValueChange() {

**this**->leftSetList->clear();

**this**->rightSetList->clear();

**this**->intersectionSetList->clear();

int i = 0;

i = 0;

**for** (auto el : \***this**->leftSet) {

**this**->leftSetList->insertItem(i++, el);

}

i = 0;

**for** (auto el : \***this**->rightSet) {

**this**->rightSetList->insertItem(i++, el);

}

i = 0;

**for** (auto el : **this**->leftSet->intersection(**this**->rightSet)) {

**this**->intersectionSetList->insertItem(i++, el);

}

}

void Widget::onClear() {

**this**->leftSet->erase();

**this**->rightSet->erase();

emit valueChanged();

}

void Widget::onOutput() {

vector<SetElementType> a;

vector<SetElementType> b;

char \*str1 = "hello";

char \*str2 = "world";

char \*str3 = "gg";

CustomString cstr1{str1};

CustomString cstr2{str2};

CustomString cstr3{str3};

a.push\_back(cstr1);

a.push\_back(cstr2);

a.push\_back(cstr3);

b.push\_back(cstr1);

b.push\_back(cstr2);

**this**->leftSet = **new** CustomSet<SetElementType>(a);

**this**->rightSet = **new** CustomSet<SetElementType>(b);

emit valueChanged();

}

Widget::Widget(QWidget \*parent) : QWidget(parent) {

auto \*mainLayout = **new** QGridLayout;

**this**->outputButton = **new** QPushButton("Print output");

**this**->clearButton = **new** QPushButton("Clear");

**this**->leftSetList = **new** QListWidget;

**this**->rightSetList = **new** QListWidget;

**this**->intersectionSetList = **new** QListWidget;

mainLayout->addWidget(**this**->leftSetList, 0, 0, 2, 1);

mainLayout->addWidget(**this**->rightSetList, 0, 1, 2, 1);

mainLayout->addWidget(**this**->intersectionSetList, 2, 0, 1, 2);

mainLayout->addWidget(**this**->outputButton, 3, 0);

mainLayout->addWidget(**this**->clearButton, 3, 1);

connect(**this**->outputButton, &QPushButton::released, **this**, &Widget::onOutput);

connect(**this**->clearButton, &QPushButton::released, **this**, &Widget::onClear);

connect(**this**, &Widget::valueChanged, &Widget::onValueChange);

setLayout(mainLayout);

}

widget.h

#pragma once

#include "custom-set.h"

#include "custom-string.h"

#include <QGridLayout>

#include <QListWidget>

#include <QPushButton>

#include <QWidget>

class Widget : public QWidget {

Q\_OBJECT

**using** SetElementType = CustomString;

signals:

void valueChanged();

public:

Widget(QWidget \*parent = nullptr);

private slots:

void onClear();

void onOutput();

void onValueChange();

private:

CustomSet<SetElementType> \*leftSet;

CustomSet<SetElementType> \*rightSet;

QPushButton \*outputButton;

QPushButton \*clearButton;

QListWidget \*leftSetList;

QListWidget \*rightSetList;

QListWidget \*intersectionSetList;

};

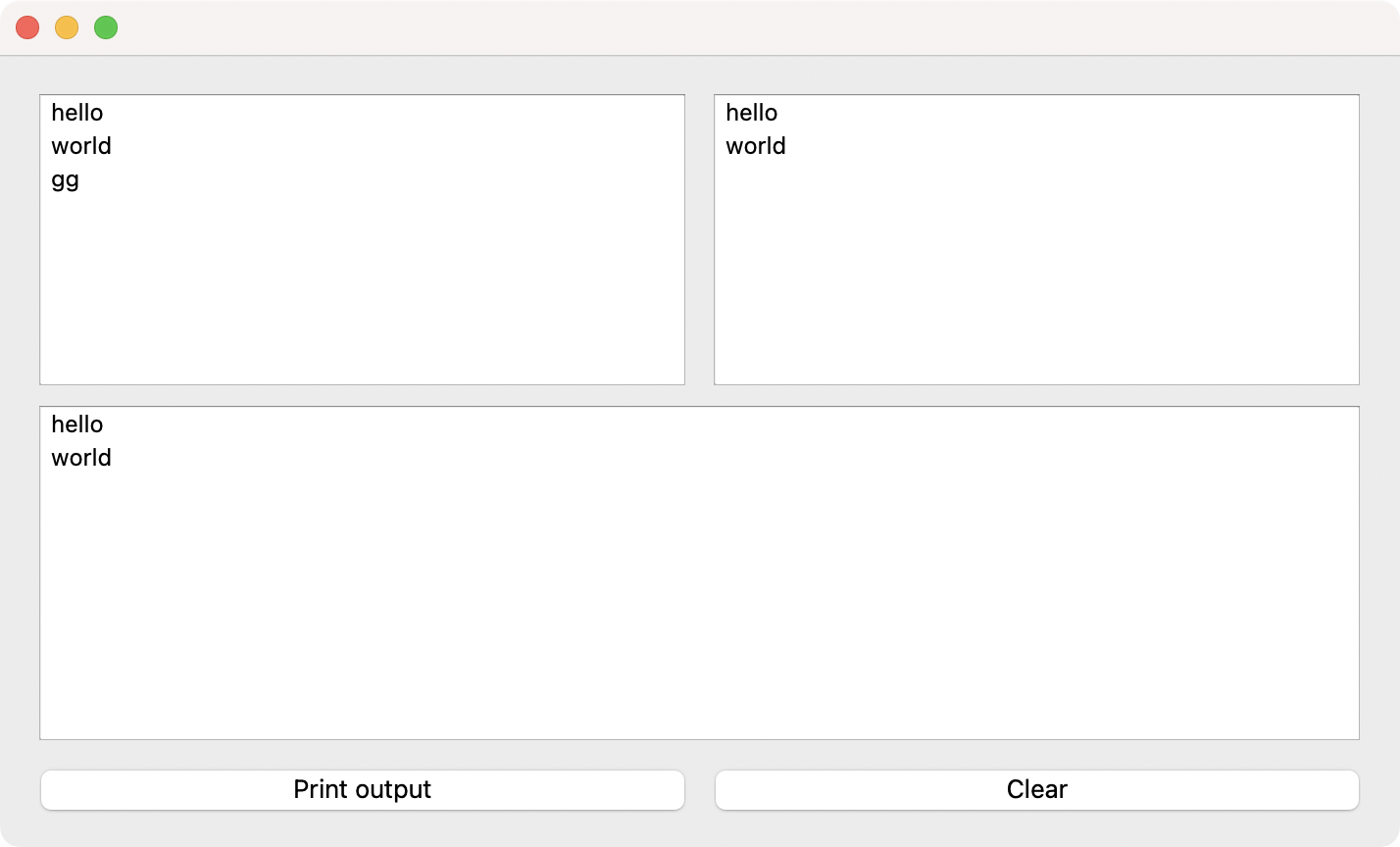
**Результат:**

Рис.1. Виконання программи

**Висновок:**

##### У ході лабораторної роботи №10 я навчився створювати шаблони класу та екземпляри їх екземпляри на прикладі шаблону класу CustomSet, що використовує вбудовані типи та користувацький CustomString.