МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №5**

по курсу “Объектно-ориентированное программирование» 1 семестр, 2021/22 уч. год

Студент: *Колпакова Диана Саргаевна, группа М8О-208Б-20*

Преподаватель: *Дорохов Евгений Павлович*

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий **одну фигуру (колонка фигура 1)**, согласно вариантам задания. Классы должны удовлетворять следующим правилам:

* + Требования к классу фигуры аналогичны требованиям из лабораторной работы No1;
  + Требования к классу контейнера аналогичны требованиям из лабораторной работы No2;
  + Класс-контейнер должен содержать объекты используя std::shared\_ptr<...>.

Нельзя использовать:

* + Стандартные контейнеры std;
  + Шаблоны (template);
  + Объекты «по-значению».

Программа должна позволять:

* + Вводить произвольное количество фигур и добавлять их в контейнер;
  + Распечатывать содержимое контейнера;
  + Удалять фигуры из контейнера.

***Вариант 9:***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Фигура №1** | **Имя класса** | **Фигура №2** | **Имя класса** | **Фигура №3** | **Имя класса** |
| Треугольник | Triangle | Квадрат | Square | Прямоугольник | Rectangle |

**Описание программы**

Исходный код лежит в 13 файлах:

1. main.cpp: часть программы, отвечающая за взаимодействие с пользователем через консоль. В ней происходит инициализация объектов и вызов функций работы с ними, заполнение стандартного контейнера вектор введенными объектами и печать его содержимого;
2. point.h: описание класса Point точек A(a1, a2);
3. point.cpp: реализация класса Point;
4. figure.h: описание абстрактного класса-родителя Figure;
5. figure.cpp: реализация класса Figure;
6. triangle.h: описание класса Triangle треугольников, заданных по трем точкам, наследника Figure;
7. triangle.cpp: реализация класса Triangle;
8. TLinkedList.h: описание класса связного списка
9. TLinkedList.cpp: реализация класса связного списка
10. TLinkedListItem.h: описание класса элемента связного списка
11. TLinkedListItem.cpp: реализация класса элемента связного списка
12. TItterator.h: описание класса итератора
13. TItterator.cpp: реализация класса итератора

Также используется файл CMakeLists.txt с конфигурацией CMake для автоматизации сборки программы.

**Дневник отладки**

Проблем не возникло

**Вывод**

Я закрепила навыки работы с шаблонами классов и научилась строить итераторы для динамических структур данных. Итераторы позволяют легко реализовать обход всех элементов некоторой структуры данных.  
Не уверена, что буду часто сталкиваться на работе с потребностью реализовать свой итератор. Но для общего развития эти знания, наверное, пригодятся.

**Исходный код**

main.cpp:

*// OOP, Lab 5 variant 9, Diana Kolpakova  
// Triangle, TLinkedList, itterator*#include <iostream>  
#include <memory>  
  
#include "triangle.h"  
#include "tlinkedlist.h"  
#include "titterator.h"  
  
using namespace std;  
  
int main()  
{  
 cout.setf(ios\_base::boolalpha);  
 cout << "oop\_exercise\_5 (c) Diana Kolpakova" << endl;  
 cout << "Triangles, TLinkedList, itterator" << endl;  
  
 shared\_ptr<TLinkedList<Triangle>> pList(new TLinkedList<Triangle>());  
  
 for (;;)  
 {  
 cout << endl;  
 cout << "Select an action for the linked list of triangles" << endl;  
 cout << "1) Is the list empty?" << endl;  
 cout << "2) Get number of triangles in the list" << endl;  
 cout << "3) Show the first triangle from the list" << endl;  
 cout << "4) Show the last triangle from the list" << endl;  
 cout << "5) Show the triangle at a specified position in the list" << endl;  
 cout << "6) Show areas of all triangles in the list" << endl;  
 cout << "7) Add a new triangle to the beginning of the list" << endl;  
 cout << "8) Add a new triangle to the end of the list" << endl;  
 cout << "9) Add a new triangle to a specified position in the list" << endl;  
 cout << "a) Remove the new first triangle from the list" << endl;  
 cout << "b) Remove the new last triangle from the list" << endl;  
 cout << "c) Remove the triangle at a specified position in the list" << endl;  
 cout << "d) Remove all triangles from the list" << endl;  
 cout << "x) End the program" << endl;  
  
 try  
 {  
 shared\_ptr<Triangle> pTriangle;  
 size\_t position;  
  
 char ch;  
 cin >> ch;  
 switch (ch)  
 {  
 case '1':  
 cout << "Is the list empty: " << pList->Empty() << endl;  
 break;  
 case '2':  
 cout << "Length of the list: " << pList->Length() << endl;  
 break;  
 case '3':  
 pTriangle = pList->First();  
 cout << \*pTriangle << endl;  
 break;  
 case '4':  
 pTriangle = pList->Last();  
 cout << \*pTriangle << endl;  
 break;  
 case '5':  
 cout << "Enter position in the list:";  
 cin >> position;  
 pTriangle = pList->GetItem(position);  
 cout << \*pTriangle << endl;  
 break;  
 case '6':  
 cout << "Triangle areas:" << endl;  
 if (pList->Empty())  
 {  
 cout << "Empty list" << endl;  
 }  
 else  
 {  
 cout << \*pList << endl;  
 }  
 break;  
 case '7':  
 cout << "Enter 3 points of triangle (6 numbers):";  
 pTriangle = shared\_ptr<Triangle>(new Triangle());  
 cin >> \*(pTriangle);  
 pList->InsertFirst(pTriangle);  
 cout << \*pTriangle << endl;  
 break;  
 case '8':  
 cout << "Enter 3 points of triangle (6 numbers):";  
 pTriangle = shared\_ptr<Triangle>(new Triangle());  
 cin >> \*(pTriangle);  
 pList->InsertLast(pTriangle);  
 cout << \*pTriangle << endl;  
 break;  
 case '9':  
 cout << "Enter 3 points of triangle (6 numbers):";  
 pTriangle = shared\_ptr<Triangle>(new Triangle());  
 cin >> \*(pTriangle);  
 cout << "Enter position in the list:";  
 cin >> position;  
 pList->Insert(pTriangle, position);  
 cout << \*pTriangle << endl;  
 break;  
 case 'a':  
 case 'A':  
 pList->RemoveFirst();  
 cout << "Removed the first triangle" << endl;  
 break;  
 case 'b':  
 case 'B':  
 pList->RemoveLast();  
 cout << "Removed the last triangle" << endl;  
 break;  
 case 'c':  
 case 'C':  
 cout << "Enter position in the list:";  
 cin >> position;  
 pList->Remove(position);  
 cout << "Removed the triangle at specified position" << endl;  
 break;  
 case 'd':  
 case 'D':  
 pList->Clear();  
 cout << "Removed all" << endl;  
 break;  
 case 'l':  
 case 'L':  
 cout << "Triangles:" << endl;  
 if (pList->Empty())  
 {  
 cout << "Empty list" << endl;  
 }  
 else  
 {  
 int i = 0;  
 for (auto pItem : \*pList)  
 {  
 std::cout << "#" << i << " " << \*pItem << std::endl;  
 i++;  
 }  
 }  
 break;  
 case 'q':  
 case 'Q':  
 case 'x':  
 case 'X':  
 cout << "Exiting" << endl;  
 return 0;  
 default:  
 cout << "Error: invalid action selected" << endl;  
 break;  
 }  
 }  
 catch (exception& ex)  
 {  
 cout << "Exception: " << ex.what() << endl;  
 }  
 }  
}

point.h:

#pragma once  
#include <iostream>  
  
using namespace std;  
  
class Point  
{  
private:  
 double x;  
 double y;  
  
public:  
 Point();  
 Point(double x, double y);  
  
 static double Distance (const Point& point1, const Point& point2);  
  
 friend istream& operator>>(istream& is, Point& point);  
 friend ostream& operator<<(ostream& os, Point& point);  
  
 bool operator==(const Point& other);  
};

point.cpp:

#include <cmath>  
#include "point.h"  
  
using namespace std;  
  
Point::Point()  
{  
 this->x = 0.0;  
 this->y = 0.0;  
}  
  
Point::Point(double x, double y)  
{  
 this->x = x;  
 this->y = y;  
}  
  
double Point::Distance(const Point& point1, const Point& point2)  
{  
 double dx = point1.x - point2.y;  
 double dy = point1.y - point2.y;  
 double distance = sqrt(dx \* dx + dy \* dy);  
 return distance;  
}  
  
bool Point::operator==(const Point& other)  
{  
 return (this->x == other.x)  
 && (this->y == other.y);  
}  
  
istream& operator>>(istream& is, Point& point)  
{  
 is >> point.x >> point.y;  
 return is;  
}  
  
ostream& operator<<(ostream& os, Point& point)  
{  
 os << "(" << point.x << ", " << point.y << ")";  
 return os;  
}

figure.h:

#pragma once  
#include "point.h"  
  
class Figure  
{  
public:  
 virtual size\_t VertexesNumber() = 0;  
 virtual double Area() = 0;  
};

triangle.h:

#pragma once  
#include "figure.h"  
  
class Triangle : public Figure  
{  
private:  
 Point point1;  
 Point point2;  
 Point point3;  
  
public:  
 Triangle();  
 Triangle(Point point1, Point point2, Point point3);  
 Triangle(const Triangle& other);  
  
 virtual size\_t VertexesNumber() override;  
 virtual double Area() override;  
  
 friend istream& operator>>(istream& is, Triangle& triangle);  
 friend ostream& operator<<(ostream& os, Triangle& triangle);  
  
 Triangle& operator=(const Triangle& other);  
 bool operator==(const Triangle& other);  
};

triangle.cpp:

#include <iostream>  
#include "triangle.h"  
  
using namespace std;  
  
Triangle::Triangle()  
{  
 this->point1 = Point();  
 this->point2 = Point();  
 this->point3 = Point();  
}  
  
Triangle::Triangle(Point point1, Point point2, Point point3)  
{  
 this->point1 = point1;  
 this->point2 = point2;  
 this->point3 = point3;  
}  
  
Triangle::Triangle(const Triangle& other)  
{  
 this->point1 = other.point1;  
 this->point2 = other.point2;  
 this->point3 = other.point3;  
}  
  
size\_t Triangle::VertexesNumber()  
{  
 return 3;  
}  
  
double Triangle::Area()  
{  
 double length12 = Point::Distance(point1, point2);  
 double length23 = Point::Distance(point2, point3);  
 double length31 = Point::Distance(point3, point1);  
 double semiPerimeter = (length12 + length23 + length31) / 2.0;  
 return sqrt(semiPerimeter \* (semiPerimeter - length12) \* (semiPerimeter - length23) \* (semiPerimeter - length31));  
}  
  
istream& operator>>(istream& is, Triangle& triangle)  
{  
 is >> triangle.point1 >> triangle.point2 >> triangle.point3;  
 return is;  
}  
  
ostream& operator<<(ostream& os, Triangle& triangle)  
{  
 os << "Triangle: " << triangle.point1 << ", " << triangle.point2 << ", " << triangle.point3;  
 return os;  
}  
  
Triangle& Triangle::operator=(const Triangle& other)  
{  
 this->point1 = other.point1;  
 this->point2 = other.point2;  
 this->point3 = other.point3;  
 return \*this;  
}  
  
bool Triangle::operator==(const Triangle& other)  
{  
 return (this->point1 == other.point1)  
 && (this->point2 == other.point2)  
 && (this->point3 == other.point3);  
}

TLinkedList.cpp:

#include <iostream>  
#include <memory>  
#include <stdexcept>  
#include "tlinkedlist.h"  
#include "triangle.h"  
  
template<typename T>  
TLinkedList<T>::TLinkedList()  
{  
 pFirstItem = nullptr;  
 pLastItem = nullptr;  
 length = 0;  
}  
  
template<typename T>  
TLinkedList<T>::TLinkedList(const TLinkedList<T>& other)  
{  
 pFirstItem = nullptr;  
 pLastItem = nullptr;  
 length = 0;  
  
 shared\_ptr<TLinkedListItem<T>> pCurrentItem = other.pFirstItem;  
 while (pCurrentItem != nullptr)  
 {  
 InsertLast(pCurrentItem->getValuePtr());  
 pCurrentItem = pCurrentItem->getNextItemPtr();  
 }  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedList<T>::First()  
{  
 if (Empty())  
 throw runtime\_error("Cannon get the item from empty list");  
 return pFirstItem->getValuePtr();  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedList<T>::Last()  
{  
 if (Empty())  
 throw runtime\_error("Cannon get the item from empty list");  
 return pLastItem->getValuePtr();  
}  
  
template<typename T>  
void TLinkedList<T>::InsertFirst(shared\_ptr<T> pValue)  
{  
 shared\_ptr<TLinkedListItem<T>> pNewItem(new TLinkedListItem<T>(pValue, pFirstItem));  
  
 pFirstItem = pNewItem;  
 if (Empty())  
 pLastItem = pNewItem;  
  
 length++;  
}  
  
template<typename T>  
void TLinkedList<T>::InsertLast(shared\_ptr<T> pValue)  
{  
 shared\_ptr<TLinkedListItem<T>> pNewItem(new TLinkedListItem<T>(pValue, nullptr));  
  
 if (pLastItem != nullptr)  
 pLastItem->setNextItemPtr(pNewItem);  
 pLastItem = pNewItem;  
 if (Empty())  
 pFirstItem = pNewItem;  
  
 length++;  
}  
  
template<typename T>  
void TLinkedList<T>::Insert(shared\_ptr<T> pValue, size\_t position)  
{  
 if (position == 0)  
 {  
 InsertFirst(pValue);  
 return;  
 }  
 else if (position == length)  
 {  
 InsertLast(pValue);  
 return;  
 }  
 else if (position > length)  
 throw runtime\_error("Specified poition is out of range");  
  
 int i = 0;  
 shared\_ptr<TLinkedListItem<T>> pCurrentItem = pFirstItem;  
 shared\_ptr<TLinkedListItem<T>> pPreviousItem = nullptr;  
 while (pCurrentItem != nullptr)  
 {  
 if (i == position)  
 break;  
 pPreviousItem = pCurrentItem;  
 pCurrentItem = pCurrentItem->getNextItemPtr();  
 i++;  
 }  
  
 shared\_ptr<TLinkedListItem<T>> pNewItem(new TLinkedListItem<T>(pValue, pCurrentItem));  
  
 pPreviousItem->setNextItemPtr(pNewItem);  
  
 length++;  
}  
  
template<typename T>  
void TLinkedList<T>::RemoveFirst()  
{  
 if (Empty())  
 throw runtime\_error("Cannon remove the item from empty list");  
 shared\_ptr<TLinkedListItem<T>> pNextItem = pFirstItem->getNextItemPtr();  
 pFirstItem = pNextItem;  
 length--;  
 if (Empty())  
 pLastItem = nullptr;  
}  
  
template<typename T>  
void TLinkedList<T>::RemoveLast()  
{  
 if (Empty())  
 throw runtime\_error("Cannon remove the item from empty list");  
  
 shared\_ptr<TLinkedListItem<T>> pCurrentItem = pFirstItem;  
 shared\_ptr<TLinkedListItem<T>> pPreviousItem = nullptr;  
 while (pCurrentItem != nullptr)  
 {  
 if (pCurrentItem == pLastItem)  
 break;  
 pPreviousItem = pCurrentItem;  
 pCurrentItem = pCurrentItem->getNextItemPtr();  
 }  
  
 if (pPreviousItem != nullptr)  
 pPreviousItem->setNextItemPtr(nullptr);  
 pLastItem = pPreviousItem;  
 length--;  
 if (Empty())  
 pFirstItem = nullptr;  
}  
  
template<typename T>  
void TLinkedList<T>::Remove(size\_t position)  
{  
 if (Empty())  
 throw runtime\_error("Cannon remove the item from empty list");  
 if (position == 0)  
 {  
 RemoveFirst();  
 return;  
 }  
 else if (position == length - 1)  
 {  
 RemoveLast();  
 return;  
 }  
 else if (position >= length)  
 throw runtime\_error("Specified poition is out of range");  
  
 int i = 0;  
 shared\_ptr<TLinkedListItem<T>> pCurrentItem = pFirstItem;  
 shared\_ptr<TLinkedListItem<T>> pPreviousItem = nullptr;  
 while (pCurrentItem != nullptr)  
 {  
 if (i == position)  
 break;  
 pPreviousItem = pCurrentItem;  
 pCurrentItem = pCurrentItem->getNextItemPtr();  
 i++;  
 }  
  
 pPreviousItem->setNextItemPtr(pCurrentItem->getNextItemPtr());  
 length--;  
}  
  
template<typename T>  
shared\_ptr<T> TLinkedList<T>::GetItem(size\_t position)  
{  
 if (Empty())  
 throw runtime\_error("Cannon get the item from empty list");  
 if (position >= length)  
 throw runtime\_error("Specified position is out of range");  
  
 int i = 0;  
 shared\_ptr<TLinkedListItem<T>> pCurrentItem = pFirstItem;  
 while (pCurrentItem != nullptr)  
 {  
 if (i == position)  
 return pCurrentItem->getValuePtr();  
 pCurrentItem = pCurrentItem->getNextItemPtr();  
 i++;  
 }  
  
 throw runtime\_error("Something went wrong");  
}  
  
template<typename T>  
bool TLinkedList<T>::Empty()  
{  
 return length == 0;  
}  
  
template<typename T>  
size\_t TLinkedList<T>::Length()  
{  
 return length;  
}  
  
template<typename T>  
void TLinkedList<T>::Clear()  
{  
 shared\_ptr<TLinkedListItem<T>> pCurrentItem = pFirstItem;  
 while (pCurrentItem != nullptr)  
 {  
 shared\_ptr<TLinkedListItem<T>> pNextItem = pCurrentItem->getNextItemPtr();  
 pCurrentItem = pNextItem;  
 }  
 pFirstItem = nullptr;  
 pLastItem = nullptr;  
 length = 0;  
}  
  
template<typename T>  
TLinkedList<T>::~TLinkedList()  
{  
 Clear();  
}  
  
template<typename T2>  
ostream& operator<<(ostream& os, const TLinkedList<T2>& list)  
{  
 shared\_ptr<TLinkedListItem<T2>> pCurrentItem = list.pFirstItem;  
 while (pCurrentItem != nullptr)  
 {  
 os << pCurrentItem->getValuePtr()->Area();  
 if (pCurrentItem != list.pLastItem)  
 os << " -> ";  
 pCurrentItem = pCurrentItem->getNextItemPtr();  
 }  
 return os;  
}  
  
template<typename T>  
TIterator<TLinkedListItem<T>, T> TLinkedList<T>::begin()  
{  
 return TIterator<TLinkedListItem<T>, T>(pFirstItem);  
}  
  
template<typename T>  
TIterator<TLinkedListItem<T>, T> TLinkedList<T>::end()  
{  
 return TIterator<TLinkedListItem<T>, T>(nullptr);  
}  
  
template class TLinkedList<Triangle>;  
template ostream& operator<<(ostream& os, const TLinkedList<Triangle>& list);

TLinkedList.h:

#pragma once  
#include <iostream>   
#include <memory>  
#include "tlinkedlistitem.h"  
#include "titterator.h"  
  
using namespace std;  
  
template <class T>  
class TLinkedList  
{  
private:  
 size\_t length;  
 shared\_ptr<TLinkedListItem<T>> pFirstItem;  
 shared\_ptr<TLinkedListItem<T>> pLastItem;  
  
public:  
 TLinkedList();  
 TLinkedList(const TLinkedList& other);  
 virtual ~TLinkedList();  
  
 shared\_ptr<T> First();  
 shared\_ptr<T> Last();  
 shared\_ptr<T> GetItem(size\_t position);  
  
 void InsertFirst(shared\_ptr<T> pValue);  
 void InsertLast(shared\_ptr<T> pValue);  
 void Insert(shared\_ptr<T> pValue, size\_t position);  
  
 void RemoveFirst();  
 void RemoveLast();  
 void Remove(size\_t position);  
  
 void Clear();  
 bool Empty();  
 size\_t Length();  
  
 template <class T2>  
 friend std::ostream& operator<<(std::ostream& os, const TLinkedList<T2>& list);  
  
 TIterator<TLinkedListItem<T>, T> begin();  
 TIterator<TLinkedListItem<T>, T> end();  
};

TItterator.h:

#pragma once  
#include <memory>  
  
using namespace std;  
  
template<typename node, typename T>  
class TIterator {  
private:  
 shared\_ptr<node> ptr;  
  
public:  
 TIterator(shared\_ptr<node> ptr)   
 {  
 this->ptr = ptr;  
 }  
  
 shared\_ptr<T> operator\*()   
 {  
 return ptr->getValuePtr();  
 }  
  
 shared\_ptr<T> operator->()   
 {  
 return ptr->getValuePtr();  
 }  
  
 TIterator<node, T> operator++()  
 {  
 return ptr = ptr->getNextItemPtr();  
 }  
  
 TIterator<node, T> operator++(int)  
 {  
 TIterator iter(\*this);  
 ++(\*this);  
 return iter;  
 }  
  
 bool operator==(TIterator<node, T> const& other)   
 {  
 return ptr == other.ptr;  
 }  
  
 bool operator!=(TIterator<node, T> const& other)   
 {  
 return !(\*this == other);  
 }  
};

TLinkedListItem.h:

#pragma once  
#include <memory>  
  
using namespace std;  
  
template <class T>  
class TLinkedListItem  
{  
private:  
 shared\_ptr<T> pValue;  
 shared\_ptr<TLinkedListItem<T>> pNextItem;  
  
public:  
 TLinkedListItem() {}  
 TLinkedListItem(shared\_ptr<T> pValue, shared\_ptr<TLinkedListItem<T>> pNextItem);  
 virtual ~TLinkedListItem();  
 shared\_ptr<T> getValuePtr();  
 shared\_ptr<TLinkedListItem<T>> getNextItemPtr();  
 void setNextItemPtr(shared\_ptr<TLinkedListItem<T>> pNextItem);  
};

TLinkedListItem.cpp:

#include <memory>  
#include "triangle.h"  
#include "tlinkedlistitem.h"  
  
using namespace std;  
  
template<class T>  
inline TLinkedListItem<T>::TLinkedListItem(shared\_ptr<T> pValue, shared\_ptr<TLinkedListItem<T>> pNextItem)  
{  
 this->pValue = pValue;  
 this->pNextItem = pNextItem;  
}  
  
template<class T>  
TLinkedListItem<T>::~TLinkedListItem()  
{  
}  
  
template<class T>  
shared\_ptr<T> TLinkedListItem<T>::getValuePtr()  
{  
 return pValue;  
}  
  
template<class T>  
shared\_ptr<TLinkedListItem<T>> TLinkedListItem<T>::getNextItemPtr()  
{  
 return pNextItem;  
}  
  
template<class T>  
void TLinkedListItem<T>::setNextItemPtr(shared\_ptr<TLinkedListItem<T>> pNextItem)  
{  
 this->pNextItem = pNextItem;  
}  
  
template class TLinkedListItem<Triangle>;

CMakeLists.txt:

cmake\_minimum\_required(VERSION 3.21)  
project(oop\_exercise\_5)  
  
set(CMAKE\_CXX\_STANDARD 14)  
  
include\_directories(.)  
  
add\_executable(oop\_exercise\_5  
 figure.cpp  
 figure.h  
 main.cpp  
 point.cpp  
 point.h  
 titterator.cpp  
 titterator.h  
 tlinkedlist.cpp  
 tlinkedlist.h  
 tlinkedlistitem.cpp  
 tlinkedlistitem.h  
 triangle.cpp  
 triangle.h)