МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №7

по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Маринин Иван Сергеевич, группа М80-208Б-20*
Преподаватель *Дорохов Евгений Павлович*

Условие

Задание: Вариант 10: Трапеция, Квадрат, Прямоугольник. Необходимо спроектировать и запрограммировать на языке C++ классы трех фигур, согласно варианту задания. Классы должны удовлетворять следующим правилам:

1. Требования к классу фигуры аналогичны требованиям из лабораторной работы №1.

2. Требования к классу фигуры аналогичны требованиям из лабораторной работы №2.

3. Класс-контейнер должен содержать объекты используя std::shared_ptr<...>.

Программа должна позволять:

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

Описание программы

Исходный файл лежит в 11 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством комманд из меню

2. figure.h: описание абстрактного класса фигур

3. point.h: описание класса точки

4. hexagon.h: описание класса шестиугольника, наследующегося от figures

5. hlist_item.h: описание класса элемента связанного списка

6. tlinkedlist.h: описание класса связанного списка

7. point.cpp: реализация класса точки

8. hexagon.cpp: реализация класса шестиугольника, наследующегося от figures

9. hlist_item.inl: реализация класса элемента связанного списка

10. tlinkedlist.inl: реализация класса связанного списка
11. iterator.h : реализация итератора

Дневник отладки

Возникли небольшие проблемы при отладке программы. После тестирования они были устранены.

```
ivanmarinin@MacBook-Air-Ivan lab_2 % ./a.out
Square List created
Print Square List
(1; 2)(1; 3)(2; 3)(2; 2) , (11; 12)(11; 13)(12; 13)(12; 12) , (21; 22)(21; 23)(22; 23)(22; 22) , (31; 32)(31; 33)
(32; 33)(32; 32)
3
1
(2; 3)(2; 4)(3; 4)(3; 3)
(21; 22)(21; 23)(22; 23)(22; 22)
(1; 1)(1; 2)(2; 2)(2; 1)
Print Square List
(2; 3)(2; 4)(3; 4)(3; 3) , (11; 12)(11; 13)(12; 13)(12; 12) , (21; 22)(21; 23)(22; 23)(22; 22)
```

Недочёты

Недочётов не было обнаружено.

Выводы

В ходе лабораторной работы №7 я ознакомился с понятием итераторов в языке C++, а также отточил навыки их использования.

Исходный код

hlist_item.inl

```
#include <iostream>
#include "hlist_item.h"

template <class T> HListItem<T>::HListItem(const std::shared_ptr<Square>
&square){
    this->square = square;
    this->next = nullptr;
}

template <class T> std::shared_ptr<HListItem<T>>
HListItem<T>::SetNext(std::shared_ptr<HListItem<T>> &next_){
    std::shared_ptr<HListItem<T>> prev = this->next;
    this->next = next_;
    return prev;
}

template <class T> std::shared_ptr<T>& HListItem<T>::GetValue(){
    return this->square;
}

template <class T> std::shared_ptr<HListItem<T>> HListItem<T>::GetNext(){
    return this->next;
}

template <class A> std::ostream& operator<<(std::ostream& os,HListItem<A> &obj)
{
    os << "[" << obj.square << "]" << std::endl;
    return os;
}

template <class T> HListItem<T>::~HListItem() {}
```

point.cpp

```
#include "point.h"
#include <cmath>

Point::Point() : x_(0.0), y_(0.0) {}

Point::Point(double x, double y) : x_(x), y_(y) {}

Point::Point(istream &is) {
  is >> x_ >> y_;
}

double Point::dist(Point& other) {
  double dx = (other.x_ - x_);
  double dy = (other.y_ - y_);
  return sqrt(dx*dx + dy*dy);
}
```

```
istream& operator>>(istream& is, Point& p) {
  is >> p.x_ >> p.y_;
  return is;
}

ostream& operator<<(ostream& os, Point& p) {
  os << "(" << p.x_ << ", " << p.y_ << ")";
  return os;
}

double Point::x(){
  return x_;
}

double Point::y(){
  return y_;
}
```

square.cpp

```
#include <iostream>
#include "square.h"

Square::Square(): a(0,0),b(0,0),c(0,0),d(0,0) {}

Square::Square(istream &is) {
  is >> a;
  is >> b;
  is >> c;
  is >> d;
}

Square::Square(Point a1, Point b1,Point c1, Point d1): a(a1),b(b1),c(c1),d(d1)
{}

double Square::Area() {
  return pow(abs(a.x() - d.x()), 2);
}

Square::~Square() {}

size_t Square::VertexesNumber() {
  return 4;
}

Square::Square(shared_ptr<Square>& other):Square(other->a, other->b, other->c,
other->d) {}

Square& Square::operator=(const Square& other) {
  if (this == &other) return *this;
  a = other.a;
  b = other.b;
  c = other.c;
  d = other.d;
  return *this;
}
```

```
Square& Square::operator == (const Square& other) {
  if (this == &other){
    cout << "Square are equal" << endl;
  }
  else {
    cout << "Square are not equal" << endl;
  }
}

ostream& operator<<(ostream& os, shared_ptr<Square>& h) {
  os << h->a << h->b << h->c << h->d;
  return os;
}
```

tlinkedlist.inl

```
#include <iostream>
#include "tlinkedlist.h"

template <class T>
Titerator<HListItem<T>, T> TLinkedList<T>::begin() {
    return Titerator<HListItem<T>, T> (front);
}

template <class T>
Titerator<HListItem<T>, T> TLinkedList<T>::end() {
    return Titerator<HListItem<T>, T>(back);
}


template <class T> TLinkedList<T>::TLinkedList() {
    size_of_list = 0;
    shared_ptr<HListItem<T>> front = nullptr;
    shared_ptr<HListItem<T>> back = nullptr;
    cout << "Square List created" << endl;
}

template <class T> TLinkedList<T>::TLinkedList(const shared_ptr<TLinkedList>
&other){
    front = other->front;
    back = other->back;
}

template <class T> size_t TLinkedList<T>::Length() {
    return size_of_list;
}

template <class T> bool TLinkedList<T>::Empty() {
    return size_of_list;
}

template <class T> shared_ptr<Square>& TLinkedList<T>::GetItem(size_t idx){
    int k = 0;
    shared_ptr<HListItem<T>> obj = front;
    while (k != idx){
```

```
        k++;
        obj = obj->GetNext();
    }
    return obj->GetValue();
}

template <class T> shared_ptr<T>& TLinkedList<T>::First() {
    return front->GetValue();
}

template <class T> shared_ptr<Square>& TLinkedList<T>::Last() {
    return back->GetValue();
}

template <class T> void TLinkedList<T>::InsertLast(const shared_ptr<Square>
&&square) {
    shared_ptr<HListItem<T>> obj (new HListItem<T>(square));
    if(size_of_list == 0) {
        front = obj;
        back = obj;
        size_of_list++;
        return;
    }
    back->SetNext(obj);
    back = obj;
    obj->next = nullptr;
    size_of_list++;
}

template <class T> void TLinkedList<T>::RemoveLast() {
    if (size_of_list == 0) {
        cout << "Square does not pop_back, because the Square List is empty" <<
endl;
    }
    else {
        if (front == back) {
        RemoveFirst();
        size_of_list--;
        return;
    }
    shared_ptr<HListItem<T>> prev_del = front;
    while (prev_del->GetNext() != back) {
      prev_del = prev_del->GetNext();
    }
    prev_del->next = nullptr;
    back = prev_del;
    size_of_list--;
    }
}

template <class T> void TLinkedList<T>::InsertFirst(const shared_ptr<Square>
&&square) {
    shared_ptr<HListItem<T>> obj (new HListItem<T>(square));
    if(size_of_list == 0) {
      front = obj;
      back = obj;
    } else {
      obj->SetNext(front); // = front;
      front = obj;
```

```
    }
    size_of_list++;
}

template <class T> void TLinkedList<T>::RemoveFirst() {
    if (size_of_list == 0) {
      cout << "Square does not pop_front, because the Square List is empty" <<
endl;
    } else {
    shared_ptr<HListItem<T>> del = front;
    front = del->GetNext();
    size_of_list--;
    }
}

template <class T> void TLinkedList<T>::Insert(const shared_ptr<Square>
&&square,size_t position) {
    if (position <0) {
        cout << "Position < zero" << endl;
    }
    else if (position > size_of_list) {
        cout << " Position > size_of_list" << endl;
    }
    else {
        shared_ptr<HListItem<T>> obj (new HListItem<T>(square));
        if (position == 0) {
        front = obj;
        back = obj;
        }
        else {
            int k = 0;
            shared_ptr<HListItem<T>> prev_insert = front;
            shared_ptr<HListItem<T>> next_insert;

            while(k+1 != position) {
            k++;
            prev_insert = prev_insert->GetNext();
            }

            next_insert = prev_insert->GetNext();
            prev_insert->SetNext(obj); // = obj;
            obj->SetNext(next_insert); // = next_insert;
        }
        size_of_list++;
  }
}

template <class T> void TLinkedList<T>::Remove(size_t position) {
    if (position > size_of_list ) {
         cout << "Position " << position << " > " << "size " << size_of_list <<
" Not correct erase" << endl;
    }
    else if (position < 0) {
        cout << "Position < 0" << endl;
    }
    else {
        if (position == 0) {
        RemoveFirst();
    }
```

```
    else {
        int k = 0;
        shared_ptr<HListItem<T>> prev_erase = front;
        shared_ptr<HListItem<T>> next_erase;
        shared_ptr<HListItem<T>> del;
        while( k+1 != position) {
            k++;
            prev_erase = prev_erase->GetNext();
        }
        next_erase = prev_erase->GetNext();
        del = prev_erase->GetNext();
        next_erase = del->GetNext();
        prev_erase->SetNext(next_erase);
    }
    size_of_list--;
  }
}

template <class T> void TLinkedList<T>::Clear() {
    shared_ptr<HListItem<T>> del = front;
    shared_ptr<HListItem<T>> prev_del;
    if(size_of_list !=0 ) {
        while(del->GetNext() != nullptr) {
            prev_del = del;
            del = del->GetNext();
        }
        size_of_list = 0;
    }
    size_of_list = 0;
    shared_ptr<HListItem<T>> front;
    shared_ptr<HListItem<T>> back;
}

template <class T> ostream& operator<<(ostream& os, TLinkedList<T>& hl) {
    if (hl.size_of_list == 0) {
        os << "The square list is empty, so there is nothing to output" <<
endl;
    }
    else {
        shared_ptr<HListItem<T>> obj = hl.front;
        os << "Print Square List" << endl;
        while(obj != nullptr) {
            if (obj->GetNext() != nullptr) {
                os << obj->GetValue() << " " << "," << " ";
                obj = obj->GetNext();
            }
            else {
                os << obj->GetValue();
                obj = obj->GetNext();
          }
        }
        os << endl;
    }
  return os;
}

template <class T> TLinkedList<T>::~TLinkedList() {
    shared_ptr<HListItem<T>> del = front;
    shared_ptr<HListItem<T>> prev_del;
```

```
    if(size_of_list !=0 ) {
        while(del->GetNext() != nullptr) {
            prev_del = del;
            del = del->GetNext();
        }
    size_of_list = 0;
    cout << "Square List deleted" << endl;
  }
}
```

main.cpp

```
#include <iostream>
#include "tlinkedlist.h"

int main() {

  TLinkedList<Square> tlinkedlist;
  cout << tlinkedlist.Empty() << endl;
  tlinkedlist.InsertLast(shared_ptr<Square>(new
Square(Point(1,2),Point(1,3),Point(3,3),Point(3,2))));
  tlinkedlist.InsertLast(shared_ptr<Square>(new
Square(Point(11,12),Point(11,13),Point(13,13),Point(13,12))));
  tlinkedlist.InsertLast(shared_ptr<Square>(new
Square(Point(21,22),Point(21,23),Point(22,23),Point(22,22))));
  tlinkedlist.InsertLast(shared_ptr<Square>(new
Square(Point(31,32),Point(31,33),Point(32,33),Point(32,32))));
  cout << tlinkedlist;
  tlinkedlist.RemoveLast();
  cout << tlinkedlist.Length() << endl;
  tlinkedlist.RemoveFirst();
  tlinkedlist.InsertFirst(shared_ptr<Square>(new
Square(Point(2,3),Point(2,4),Point(3,4),Point(3,3))));
  tlinkedlist.Insert(shared_ptr<Square>(new
Square(Point(1,1),Point(1,2),Point(2,2),Point(2, 1))),2);
  cout << tlinkedlist.Empty() << endl;
  cout << tlinkedlist.First() << endl;
  cout << tlinkedlist.Last() << endl;
  cout << tlinkedlist.GetItem(2) << endl;
  tlinkedlist.Remove(2);
  cout << tlinkedlist;
  tlinkedlist.Clear();
  return 0;
}
```

iterator.h

```
#ifndef ITEATOR_H
#define ITEATOR_H

template <class node, class T> class Titerator {
public:
    Titerator(shared_ptr<node> n) {
```

```
        node_ptr = n;
    }

    shared_ptr<T> operator*() {
        return node_ptr->GetValue();
    }
    shared_ptr<T> operator->() {
        return node_ptr->GetValue();
    }

    void operator++() {
        node_ptr = node_ptr->GetNext();
    }
    Titerator operator++(int) {
        Titerator other(*this);
        ++(*this);
        return other;
    }

    bool operator==(Titerator const &i) {
        return node_ptr == i.node_ptr;
    };
    bool operator!=(Titerator const &i) {
        return node_ptr != i.node_ptr;
    };

private:
    shared_ptr<node> node_ptr;
};

#endif
```