

Лабораторная работа №4

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Медведев Данила Андреевич, М80-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

Задание:

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

- Требования к классам фигуры аналогичны требованиям из лабораторной работы №1;
- Требования к классу контейнера аналогичны требованиям из лабораторной работы №2;
- Шаблон класса-контейнера должен содержать объекты используя `std::shared_ptr`.

Нельзя использовать:

- Стандартные контейнеры `std`.

Программа должна позволять:

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

Программа должна позволять:

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

Вариант №11

- Фигура 1: Прямоугольник (Rectangle)
- Структура: Связный список

Описание программы:

Исходный код разделён на несколько файлов:

- `point.h(cpp)` – описание и реализация класса точки.
- `figure.h(cpp)` – описание и реализация класса фигуры.
- `rectangle.h(cpp)` – описание и реализация класса прямоугольника (наследуется от фигуры).
- `tlinkedlist.h(cpp)` - описание и реализация класса связного списка.
- `tlinkedlist_i.h(cpp)` – описание и реализация класса отдельного элемента списка.

Дневник отладки

Программа в отладке не нуждалась.

Вывод:

Проделав данную работу, я продолжил изучение базовых понятий ооп. По сути здесь я переделал предыдущую лабораторную работу, но в этот раз с использованием `templates`. `Template` помогает сделать нам шаблоны функций, в которых мы позже сможем использовать переменные типа, нужного для решения конкретной задачи. Я считаю это очень полезным, ведь при использовании шаблона нам не придется переделывать функции под каждый тип данных.

Исходный код:

Figure.h

```
#pragma once

#include <iostream>
#include "point.h"
using namespace std;

class Figure {
public:
    virtual size_t VertexesNumber() = 0;
    virtual double Area() = 0;
    virtual void Print(std::ostream& os) = 0;
protected:
    Point a;
    Point b;
    Point c;
    Point d;
};
```

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

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

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

double Point::getY()
{
    return y_;
}

void Point::setX(double a)
{
    x_ = a;
}

void Point::setY(double a)
{
    y_ = a;
}

std::istream& operator>>(std::istream& is, Point& p) {
```

```

        is >> p.x_ >> p.y_;
        return is;
    }

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

    bool operator== (Point& p1, Point& p2)
    {
        return (p1.getX() == p2.getY() &&
                p1.getY() == p2.getY());
    }

    bool operator!= (Point& p1, Point& p2)
    {
        return !(p1 == p2);
    }

```

Point.h

```

#pragma
once

```

```

#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
    Point();
    Point(std::istream& is);
    Point(double x, double y);

    double dist(Point& other);
    double getX();
    double getY();
    void setX(double a);
    void setY(double a);

    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os,const Point& p);

    friend bool operator== (Point& p1, Point& p2);
    friend bool operator!= (Point& p1, Point& p2);

private:
    double x_;
    double y_;
};

#endif

```

Rectangle.cpp

```

#include
<iostream>

#include"point.h"

```

```

#include "rectangle.h"

using namespace std;

Rectangle::Rectangle(Point a1, Point a2, Point a3, Point a4) {

    a = a1;

    b = a2;

    c = a3;

    d = a4;

}

Rectangle::Rectangle() {

    a.setX(0);

    a.setY(0);

    b.setX(0);

    b.setY(0);

    c.setX(0);

    c.setY(0);

    d.setX(0);

    d.setY(0);

}

double Rectangle::Area() {

    double A = a.dist(b);

    double B = b.dist(c);

    return A * B;

}

void Rectangle::Print(std::ostream& os)

{

    std::cout << "Rectangle: " << a << " " << b << " " << c << " " << d << endl;

}

size_t Rectangle::VertexesNumber()

{

    return (size_t)4;

```

```

    }

    Rectangle::Rectangle(std::istream& is) {

        cin >> a >> b >> c >> d;

    }

    std::istream& operator>>(std::istream& is, Rectangle& p) {

        is >> p.a >> p.b >> p.c >> p.d;

        return is;

    }

    std::ostream& operator<<(std::ostream& os, const Rectangle& p) {

        os << p.a << " " << p.b << " " << p.c << " " << p.d;

        return os;

    }

    bool operator== (Rectangle& p1, Rectangle& p2)

    {

        return (p1.a == p2.a &&

                p1.b == p2.b && p1.c == p2.c && p1.d == p2.d);

    }

    bool operator!= (Rectangle& p1, Rectangle& p2)

    {

        return !(p1 == p2);

    }

```

Rectangle.h

```

#pragma
once

#include <iostream>
#include "point.h"
#include "figure.h"
class Rectangle : Figure {
public:

    double Area();

    void Print(std::ostream& os);

    size_t VerticesNumber();

```

```

        Rectangle(Point a1, Point a2, Point a3, Point a4);
        Rectangle(std::istream& is);
        Rectangle();
        friend std::istream& operator>>(std::istream& is, Rectangle& p);
        friend std::ostream& operator<<(std::ostream& os, const Rectangle& p);

        friend bool operator== (Rectangle& r1, Rectangle& r2);
        friend bool operator!= (Rectangle& r1, Rectangle& r2);

private:

};

```

TLinkedList.cpp

```

#include
"tLinkedList.h"

```

```

template<typename T>
TLinkedList<T>::TLinkedList() {
    len = 0;
    head = nullptr;
}

template<typename T>
TLinkedList<T>::TLinkedList(const TLinkedList<T>& list) {
    len = list.len;
    if (!list.len) {
        head = nullptr;
        return;
    }
    head = make_shared<TLinkedListItem<T>>(list.head->GetVal(), nullptr);
    shared_ptr<TLinkedListItem<T>> cur = head;
    shared_ptr<TLinkedListItem<T>> it = list.head;
    for (size_t i = 0; i < len - 1; ++i) {
        it = it->GetNext();
        shared_ptr<TLinkedListItem<T>> new_item = make_shared<TLinkedListItem<T>>(it->GetVal(), nullptr);
        cur->SetNext(new_item);
        cur = cur->GetNext();
    }
}

template<typename T>
shared_ptr<T> TLinkedList<T>::First() {
    if (len == 0) {
        return nullptr;
    }
    return head->GetVal();
}

template<typename T>
shared_ptr<T> TLinkedList<T>::Last() {
    if (len == 0) {
        return nullptr;
    }
    shared_ptr<TLinkedListItem<T>> cur = head;
    for (size_t i = 0; i < len - 1; ++i) {
        cur = cur->GetNext();
    }
    return cur->GetVal();
}

template<typename T>
void TLinkedList<T>::InsertFirst(shared_ptr<T> figure) {
    shared_ptr<TLinkedListItem<T>> it = make_shared<TLinkedListItem<T>>(figure, head);
    head = it;
}

```

```

        len++;
    }

template<typename T>
void TLinkedList<T>::InsertLast(shared_ptr<T> figure) {
    if (len == 0) {
        head = make_shared<TLinkedListItem<T>>(figure, nullptr);
        len = 1;
        return;
    }
    shared_ptr<TLinkedListItem<T>> cur = head;
    for (size_t i = 0; i < len - 1; ++i) {
        cur = cur->GetNext();
    }
    shared_ptr<TLinkedListItem<T>> it = make_shared<TLinkedListItem<T>>(figure, nullptr);
    cur->SetNext(it);
    len++;
}

template<typename T>
void TLinkedList<T>::Insert(shared_ptr<T> figure, size_t pos) {
    if (pos > len || pos < 0) {
        return;
    }
    shared_ptr<TLinkedListItem<T>> cur = head;
    shared_ptr<TLinkedListItem<T>> prev = nullptr;
    for (size_t i = 0; i < pos; ++i) {
        prev = cur;
        cur = cur->GetNext();
    }
    shared_ptr<TLinkedListItem<T>> it = make_shared<TLinkedListItem<T>>(figure, cur);
    if (prev) {
        prev->SetNext(it);
    }
    else {
        head = it;
    }
    len++;
}

template<typename T>
void TLinkedList<T>::RemoveFirst() {
    if (!len) return;
    shared_ptr<TLinkedListItem<T>> del = head;
    head = head->GetNext();
    len--;
}

template<typename T>
void TLinkedList<T>::RemoveLast() {
    if (!len) return;
    if (len == 1) {
        head = nullptr;
        len = 0;
        return;
    }
    shared_ptr<TLinkedListItem<T>> cur = head;
    for (size_t i = 0; i < len - 2; ++i) {
        cur = cur->GetNext();
    }
    shared_ptr<TLinkedListItem<T>> del = cur->GetNext();
    cur->SetNext(nullptr);
    len--;
}

template<typename T>

```



```

void TLinkedList<T>::Remove(size_t pos) {
    if (!len) return;
    if (pos < 0 || pos >= len) return;
    shared_ptr<TLinkedListItem<T>> cur = head;
    shared_ptr<TLinkedListItem<T>> prev = nullptr;
    for (size_t i = 0; i < pos; ++i) {
        prev = cur;
        cur = cur->GetNext();
    }
    if (prev) {
        prev->SetNext(cur->GetNext());
    }
    else {
        head = cur->GetNext();
    }
    len--;
}

template<typename T>
shared_ptr<T> TLinkedList<T>::GetItem(size_t ind) {
    if (ind < 0 || ind >= len) return nullptr;
    shared_ptr<TLinkedListItem<T>> cur = head;
    for (size_t i = 0; i < ind; ++i) {
        cur = cur->GetNext();
    }
    return cur->GetVal();
}

template<typename T>
bool TLinkedList<T>::Empty() {
    return len == 0;
}

template<typename T>
size_t TLinkedList<T>::Length() {
    return len;
}

template<typename T>
std::ostream& operator<<(std::ostream& os, const TLinkedList<T>& list) {
    shared_ptr<TLinkedListItem<T>> cur = list.head;
    os << "List: \n";
    for (size_t i = 0; i < list.len; ++i) {
        os << *cur;
        cur = cur->GetNext();
    }
    return os;
}

template<typename T>
void TLinkedList<T>::Clear() {
    while (!this->Empty()) {
        this->RemoveFirst();
    }
}

template<typename T>
TLinkedList<T>::~TLinkedList() {
    while (!this->Empty()) {
        this->RemoveFirst();
    }
}

template
class TLinkedList<Rectangle>;

```

```
template std::ostream& operator<<(std::ostream& os, const TLinkedList<Rectangle>& list);
```

Tlinkedlist.h

```
#pragma
once

#include "tlinkedlist_i.h"

template<typename T>
class TLinkedList {
private:
    size_t len;
    shared_ptr<TLinkedListItem<T>> head;
public:
    TLinkedList();

    TLinkedList(const TLinkedList<T>& list);

    shared_ptr<T> First();

    shared_ptr<T> Last();

    void InsertFirst(shared_ptr<T> rectangle);

    void InsertLast(shared_ptr<T> rectangle);

    void Insert(shared_ptr<T> rectangle, size_t pos);

    void RemoveFirst();

    void RemoveLast();

    void Remove(size_t pos);

    shared_ptr<T> GetItem(size_t ind);

    bool Empty();

    size_t Length();

    template<typename X>
    friend std::ostream& operator<<(std::ostream& os, const TLinkedList<X>& list);

    void Clear();

    virtual ~TLinkedList();
};
```

Tlinkedlist_i.cpp

```
#include
"tlinkedlist_i.h"

template<typename T>
TLinkedListItem<T>::TLinkedListItem(shared_ptr<T> figure, shared_ptr<TLinkedListItem<T>> nxt) {
    val = figure;
    next = nxt;
}

template<typename T>
shared_ptr<TLinkedListItem<T>> TLinkedListItem<T>::GetNext() {
    return next;
}
```

```

template<typename T>
void TLinkedListItem<T>::SetNext(shared_ptr<TLinkedListItem<T>> nxt) {
    next = nxt;
}

template<typename T>
shared_ptr<T> TLinkedListItem<T>::GetVal() {
    return val;
}

template<typename T>
std::ostream& operator<<(std::ostream& os, const TLinkedListItem<T>& item) {
    os << *item.val;
    return os;
}

template class TLinkedListItem<Rectangle>;
template std::ostream& operator<<(std::ostream& os, const TLinkedListItem<Rectangle>& item);

template<typename T>
TLinkedListItem<T>::~TLinkedListItem() {

}

```

TLinkedList_i.h

```

#pragma
once

```

```

#include "rectangle.h"
#include "iostream"
#include "memory"

using std::shared_ptr;
using std::make_shared;

template <typename T>
class TLinkedListItem {
private:
    shared_ptr<T> val;
    shared_ptr<TLinkedListItem<T>> next;
public:
    TLinkedListItem(shared_ptr<T> rectangle, shared_ptr<TLinkedListItem<T>> nxt);

    void SetNext(shared_ptr<TLinkedListItem<T>> nxt);

    shared_ptr<TLinkedListItem<T>> GetNext();

    shared_ptr<T> GetVal();

    template<typename T1>
    friend std::ostream& operator<<(std::ostream& os, const TLinkedListItem<T1>& item);

    virtual ~TLinkedListItem();
};

```