**Лабораторная работа №6**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Капичников Ярослав Андреевич, группа М8О-207Б-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**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Дата | Событие | Действие по исправлению |
| 1 |  |  |  |
|  |  |  |  |

**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 VertexesNumber(); |
|  | 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(); |
|  | }; |