МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №7**

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

Студентка: *Волошинская Евгения Владимировна, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович*

**Задание**

Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for. Например:

for(auto i : stack) {

std::cout << \*i << std::endl;

}

Нельзя использовать:

* Стандартные контейнеры std.

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** | **Контейнер 1-го уровня** | **Имя класса** |
| Треугольник | Triangle | Связанный список | TLinkedList |

**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. item.h: описание класса Item, объектами которого являются элементы связанного списка;
9. item.cpp: реализация класса Item;
10. titerator.h: описание и реализация класса TIterator, объекты которого – итераторы, указывающие на элементы Item контейнера TLinkedList.
11. tlinkedlist.h: описание класса TLinkedList, объекты которого – связанные списки элементов типа Item;
12. tlinkedlist.cpp: реализация класса TLinkedList;
13. templates.cpp: создание экземпляров шаблонов элементов контейнера и дружественной функции вывода значения площади элемента.

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

**Дневник отладки**

***Ошибка***: In file included from templates.cpp:5:0:

tlinkedlist.cpp: In instantiation of ‘size\_t TLinkedList<T>::Length() [with T = Triangle; size\_t = long unsigned int]’:

templates.cpp:10:16: required from here

tlinkedlist.cpp:35:52: error: no ‘operator++(int)’ declared for postfix ‘++’ [-fpermissive]

for (TIterator<T> i = IFirst(); i != ILast(); i++) {

***Решение***: Реализовала перегрузку и префиксного, и постфиксного оператора инкремента. Причем для их различения у префиксного нет передаваемого аргумента, а для постфиксного надо указать аргумент типа int.

***Ошибка***: In file included from templates.cpp:5:0:

tlinkedlist.cpp: In instantiation of ‘std::ostream& operator<<(std::ostream&, const TLinkedList<T>&) [with T = Triangle; std::ostream = std::basic\_ostream<char>]’:

templates.cpp:11:86: required from here

tlinkedlist.cpp:212:23: error: passing ‘const TLinkedList<Triangle>’ as ‘this’ argument discards qualifiers [-fpermissive]

for (TIterator<T> i = list.IFirst(); i != list.ILast(); i++) {

^

tlinkedlist.cpp:20:14: note: in call to ‘TIterator<T> TLinkedList<T>::IFirst() [with T = Triangle]’

TIterator<T> TLinkedList<T>::IFirst()

^~~~~~~~~~~~~~

***Решение***: this (объект TLinkedList) при перегрузке оператора вывода не должен изменяться при передаче в функции IFirst, ILast, поэтому для них нужно указать const.

**Вывод**

В данной лабораторной работе я продолжила изучать основы ООП в языке C++. Я познакомилась с понятием итераторов, научилась создавать итераторы для динамических структур данных и использовать их при написании функций работы с элементами этих структур, закрепила навыки написания шаблонов функций и классов. Я реализовала шаблонный класс итераторов TIterator, каждый объект которого хранит ссылку на элемент контейнера. Для использования итераторов в уже написанных методах класса-контейнера TLinkedList я написала функции сравнения операторов, реализовала для них перегрузку операторов инкремента и декремента, получения элемента по итератору. Думаю, что итератор – это важное понятие в программировании. Итераторы не только универсальны, так как позволяют осуществлять доступ и перебор в любых структурах данных, но и полезны, так как повышают уровень абстракции, делая код более читаемым и позволяя избежать некоторых проблем при изменениях в исходной структуре данных.

**Исходный код**

main.cpp:

#include "tlinkedlist.h"

//#include "tlinkedlist.cpp"

int main(void)

{

TLinkedList<Triangle> l;

Point a1(-3, -1);

Point b1(3, 0);

Point c1(4, 8);

Point a2(0, 0);

Point b2(2, 3);

Point c2(-2, 6);

Point a3(1, 0);

Point b3(0.5, 1);

Point c3(2, 1);

std::shared\_ptr<Triangle> t1(new Triangle (a1, b1, c1));

std::shared\_ptr<Triangle> t2(new Triangle (a2, b2, c2));

std::shared\_ptr<Triangle> t3(new Triangle (a3, b3, c3));

std::cout << l << std::endl;

std::cout << "Length: " << l.Length() << std::endl;

l.Insert(t1, 1);

std::cout << l << std::endl;

l.Insert(t1, 3);

l.Insert(t2, 2);

std::cout << l << std::endl;

l.InsertLast(t1);

std::cout << l << std::endl;

l.Insert(t3, 4);

std::cout << l << std::endl;

std::cout << "Length: " << l.Length() << std::endl;

l.Insert(t3, 3);

std::cout << l << std::endl;

l.Insert(t2, 6);

std::cout << l << std::endl;

l.Insert(t2, 1);

std::cout << l << std::endl;

l.InsertFirst(t3);

std::cout << l << std::endl;

l.Remove(9);

l.Remove(5);

std::cout << l << std::endl;

std::cout << "Length: " << l.Length() << std::endl;

l.Remove(l.Length());

std::cout << l << std::endl;

l.RemoveFirst();

std::cout << l << std::endl;

std::cout << "Length: " << l.Length() << std::endl;

l.RemoveLast();

std::cout << l << std::endl;

l.InsertFirst(t3);

std::cout << l << std::endl;

std::cout << \*l.First() << std::endl;

std::cout << \*l.Last() << std::endl;

std::cout << "Length: " << l.Length() << std::endl;

std::cout << "GetItem result:" << std::endl;

std::cout << \*l.GetItem(1) << std::endl;

std::cout << \*l.GetItem(2) << std::endl;

std::cout << \*l.GetItem(3) << std::endl;

std::cout << \*l.GetItem(4) << std::endl;

std::cout << \*l.GetItem(5) << std::endl;

std::cout << "vs range-based for loop result:" << std::endl;

for (auto i : l) {

std::cout << \*i << std::endl;

}

l.Clear();

std::cout << l << std::endl;

std::cout << "Length: " << l.Length() << std::endl;

return 0;

}

point.h:

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

friend std::istream& operator>>(std::istream& is, Point& p);

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

public:

Point();

Point(double x, double y);

Point(std::istream &is);

bool operator==(const Point &other);

double dist(Point& other);

private:

double x\_;

double y\_;

};

#endif // POINT\_H

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\_;

}

bool Point::operator==(const Point &other)

{

return ((x\_ == other.x\_) && (y\_ == other.y\_));

}

double Point::dist(Point& other) {

double dx = (other.x\_ - x\_);

double dy = (other.y\_ - y\_);

return std::sqrt(dx\*dx + dy\*dy);

}

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;

}

figure.h:

#ifndef FIGURE\_H

#define FIGURE\_H

#include "point.h"

class Figure {

public:

virtual size\_t VertexesNumber() = 0;

virtual void Print(std::ostream& os) = 0;

virtual double Area() = 0;

virtual ~Figure() {};

};

#endif // FIGURE\_H

triangle.h:

#ifndef TRIANGLE\_H

#define TRIANGLE\_H

#include <iostream>

#include "figure.h"

class Triangle : public Figure {

public:

Triangle();

Triangle(Point a, Point b, Point c);

Triangle(std::istream &is);

Triangle(const Triangle& other);

Triangle &operator=(const Triangle &other);

bool operator==(const Triangle &other);

friend std::istream& operator>>(std::istream& is, Triangle& o);

friend std::ostream& operator<<(std::ostream& os, const Triangle& t);

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

virtual ~Triangle();

private:

Point p1;

Point p2;

Point p3;

};

#endif // TRIANGLE\_H

triangle.cpp:

#include "triangle.h"

#include <iostream>

#include <cmath>

Triangle::Triangle()

: p1(0.0, 0.0), p2(0.0, 0.0), p3(0.0, 0.0) { // можно, но длиннее p1(Point(0.0, 0.0))

//std::cout << "Default triangle created" << std::endl;

}

Triangle::Triangle(Point a, Point b, Point c)

: p1(a), p2(b), p3(c) {

//std::cout << "Triangle created by parameters" << std::endl;

}

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

is >> p1 >> p2 >> p3;

}

Triangle::Triangle(const Triangle& other)

: Triangle(other.p1, other.p2, other.p3) {

//std::cout << "Triangle copy created" << std::endl;

}

Triangle &Triangle::operator=(const Triangle &other)

{

if (this == &other) {

return \*this;

}

p1 = other.p1;

p2 = other.p2;

p3 = other.p3;

return \*this;

}

bool Triangle::operator==(const Triangle &other)

{

return (p1 == other.p1) && (p2 == other.p2) && (p3 == other.p3);

}

std::istream& operator>>(std::istream& is, Triangle& t)

{

is >> t.p1 >> t.p2 >> t.p3;

return is;

}

std::ostream& operator<<(std::ostream& os, const Triangle& t)

{

os << "Triangle: " << t.p1 << " " << t.p2 << " " << t.p3 << std::endl;

return os;

}

size\_t Triangle::VertexesNumber() {

return(size\_t)3;

}

double Triangle::Area() {

double p12 = p1.dist(p2);

double p13 = p1.dist(p3);

double p23 = p2.dist(p3);

double p = (p12 + p23 + p13) / 2.0;

return std::sqrt(p \* (p - p12) \* (p - p23) \* (p - p13));

}

void Triangle::Print(std::ostream& os) {

os << "Triangle: ";

os << p1 << ", ";

os << p2 << ", ";

os << p3 << std::endl;

}

Triangle::~Triangle() {

//std::cout << "Triangle deleted" << std::endl;

}

item.h:

#ifndef ITEM\_H

#define ITEM\_H

#include "triangle.h"

#include <memory>

template <class T>

class Item

{

public:

Item(const std::shared\_ptr<T> o);

Item(const std::shared\_ptr< Item<T> > other);

std::shared\_ptr< Item<T> > Left();

std::shared\_ptr< Item<T> > Right();

void InsLeft(std::shared\_ptr< Item<T> > item);

void InsRight(std::shared\_ptr< Item<T> > item);

std::shared\_ptr<T> GetTriangle();

template <class I>

friend std::ostream &operator<<(std::ostream &os, const std::shared\_ptr< Item<T> > item);

virtual ~Item();

private:

std::shared\_ptr<T> object;

std::shared\_ptr< Item<T> > prev;

std::shared\_ptr< Item<T> > next;

};

#endif // ITEM\_H

item.cpp:

#include "item.h"

template <class T>

Item<T>::Item(const std::shared\_ptr<T> o)

{

this->object = o;

this->next = nullptr;

this->prev = nullptr;

}

template <class T>

Item<T>::Item(const std::shared\_ptr< Item<T> > other)

{

this->object = other->object;

this->next = other->next;

this->prev = other->prev;

}

template <class T>

std::shared\_ptr< Item<T> > Item<T>::Left()

{

return this->prev;

}

template <class T>

std::shared\_ptr< Item<T> > Item<T>::Right()

{

return this->next;

}

template <class T>

void Item<T>::InsLeft(std::shared\_ptr< Item<T> > item)

{

this->prev = item;

}

template <class T>

void Item<T>::InsRight(std::shared\_ptr< Item<T> > item)

{

this->next = item;

}

template <class T>

std::shared\_ptr<T> Item<T>::GetTriangle()

{

return this->object;

}

template <class T>

std::ostream &operator<<(std::ostream &os, const std::shared\_ptr< Item<T> > item)

{

os << item->object << std::endl;

return os;

}

template <class T>

Item<T>::~Item() {}

tlinkedlist.h:

#ifndef TLINKEDLIST\_H

#define TLINKEDLIST\_H

#include "item.h"

#include "titerator.h"

template <class T>

class TLinkedList

{

public:

TLinkedList();

TLinkedList(const TLinkedList<T>& other);

size\_t Length();

bool Empty();

TIterator<T> begin() const;

TIterator<T> end() const;

const std::shared\_ptr<T> First();

const std::shared\_ptr<T> Last();

const std::shared\_ptr<T> GetItem(size\_t idx);

void InsertFirst(const std::shared\_ptr<T> object);

void InsertLast(const std::shared\_ptr<T> object);

void Insert(const std::shared\_ptr<T> object, size\_t position);

void RemoveFirst();

void RemoveLast();

void Remove(size\_t position);

template <class F> //friend function needs its own template for sharing private fields

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

void Clear();

virtual ~TLinkedList();

private:

std::shared\_ptr< Item<T> > head;

std::shared\_ptr< Item<T> > tail;

};

#endif // TLINKEDLIST\_H

tlinkedlist.cpp:

#include "tlinkedlist.h"

template <class T> //needed before all definitions of functions out of class definition

TLinkedList<T>::TLinkedList() : head(nullptr), tail(nullptr) {}

template <class T>

TLinkedList<T>::TLinkedList(const TLinkedList &other)

{

head = other.head;

tail = other.tail;

}

template <class T>

bool TLinkedList<T>::Empty()

{

return (head == nullptr);

}

template <class T>

TIterator<T> TLinkedList<T>::begin() const

{

return TIterator<T> (head);

}

template <class T>

TIterator<T> TLinkedList<T>::end() const

{

return TIterator<T> (nullptr);

}

template <class T>

size\_t TLinkedList<T>::Length()

{

size\_t size = 0;

for (auto i: \*this) {

++size;

}

/\*

for (TIterator<T> i = begin(); i != end(); i++) {

++size;

}

\*/

return size;

}

template <class T>

const std::shared\_ptr<T> TLinkedList<T>::First()

{

if (head == nullptr) {

std::cout << "List is empty" << std::endl;

exit(1);

}

return head -> GetTriangle();

}

template <class T>

const std::shared\_ptr<T> TLinkedList<T>::Last()

{

if (head == nullptr) {

std::cout << "List is empty" << std::endl;

exit(1);

}

return tail -> GetTriangle();

}

template <class T>

const std::shared\_ptr<T> TLinkedList<T>::GetItem(size\_t idx)

{

size\_t len = Length();

if (head == nullptr) {

std::cout << "List is empty" << std::endl;

exit(1);

}

if (idx > len) {

std::cout << "No element on position " << idx << std::endl;

exit(1);

}

//std::shared\_ptr< Item<T> > item = head;

TIterator<T> i = begin();

for (size\_t c = 1; c < idx; ++c) {

i++;

}

return \*i;

}

template <class T>

void TLinkedList<T>::InsertFirst(const std::shared\_ptr<T> object)

{

std::shared\_ptr<Item<T>> item(new Item<T>(object));

if (head == nullptr) {

head = item;

tail = item;

return;

} // важно, или будет обращение к nullptr -> prev

//item->InsLeft(nullptr);

item->InsRight(head);

head->InsLeft(item);

head = item;

}

template <class T>

void TLinkedList<T>::InsertLast(const std::shared\_ptr<T> object)

{

std::shared\_ptr<Item<T>> item(new Item<T>(object));

if (head == nullptr) {

head = item;

tail = item;

return;

}

tail->InsRight(item);

item->InsLeft(tail);

//item->InsRight(nullptr);

tail = item;

}

template <class T>

void TLinkedList<T>::Insert(const std::shared\_ptr<T> object, size\_t position)

{

size\_t len = Length();

if (position > len + 1) {

std::cout << "No such position" << std::endl;

return;

}

if (position == 1) {

InsertFirst(object);

return;

}

if (position == len + 1) {

InsertLast(object);

return;

}

std::shared\_ptr<Item<T>> item(new Item<T>(object));

std::shared\_ptr<Item<T>> curr = head;

for (size\_t i = 1; i < position; ++i) {

curr = curr->Right();

}

std::shared\_ptr<Item<T>> prev = curr->Left();

prev->InsRight(item);

curr->InsLeft(item);

item->InsLeft(prev);

item->InsRight(curr);

}

template <class T>

void TLinkedList<T>::RemoveFirst()

{

if (head == nullptr) {

std::cout << "List is empty" << std::endl;

return;

}

if (head == tail) {

head = nullptr;

tail = nullptr;

return;

}

std::shared\_ptr<Item<T>> item = head;

head = head->Right();

head->InsLeft(nullptr);

}

template <class T>

void TLinkedList<T>::RemoveLast()

{

if (head == nullptr) {

std::cout << "List is empty" << std::endl;

return;

}

if (head == tail) {

head = nullptr;

tail = nullptr;

return;

}

std::shared\_ptr<Item<T>> item = tail;

tail = tail->Left();

tail->InsRight(nullptr);

}

template <class T>

void TLinkedList<T>::Remove(size\_t position)

{

size\_t len = Length();

if (head == nullptr) {

std::cout << "List is empty" << std::endl;

return;

}

if (position > len) {

std::cout << "No such position" << std::endl;

return;

}

if (position == 1) {

RemoveFirst();

return;

}

if (position == len) {

RemoveLast();

return;

}

std::shared\_ptr<Item<T>> item = head;

for (size\_t i = 1; i < position; ++i) {

item = item->Right();

}

std::shared\_ptr<Item<T>> left = item->Left();

std::shared\_ptr<Item<T>> right = item->Right();

left->InsRight(right);

right->InsLeft(left);

}

template <class T> //also needed before friend

std::ostream &operator<<(std::ostream &os, const TLinkedList<T> &list)

{

if (list.head == nullptr) {

os << "List is empty";

return os;

}

for (std::shared\_ptr<Item<T>> i = list.head; i != nullptr; i = i->Right()) {

if (i->Right() != nullptr)

os << i->GetTriangle()->Area() << " -> ";

else

os << i->GetTriangle()->Area();

}

return os;

}

/\*

for (TIterator<T> i = list.begin(); i != list.end(); i++) {

if ((i++).get() != nullptr)

if ((i++) != list.end())

os << (\*i)->Area() << " -> ";

else

os << (\*i)->Area();

}

return os;

}

\*/

template <class T>

void TLinkedList<T>::Clear()

{

while (head != nullptr) {

RemoveFirst();

}

}

template <class T>

TLinkedList<T>::~TLinkedList()

{

while (head != nullptr) {

RemoveFirst();

}

}

templates.cpp:

// loading ALL files, containing templates, is essential

#include "item.h"

#include "item.cpp" //essential as well

#include "tlinkedlist.h"

#include "tlinkedlist.cpp"

//creating instances of the defined in '...'.h templates

template class Item<Triangle>;

//template class TIterator<Triangle>;

template class TLinkedList<Triangle>;

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

titerator.h:

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include "item.h"

template <class T>

class TIterator {

public:

TIterator(std::shared\_ptr< Item<T> > n)

{

node = n;

}

std::shared\_ptr<T> operator\*() const

{

return node->GetTriangle();

}

std::shared\_ptr<T> operator->() const

{

return node->GetTriangle();

}

std::shared\_ptr< Item<T> > get()

{

return node;

}

void operator++()

{

node = node->Right();

}

void operator--()

{

node = node->Left();

}

TIterator<T> operator++(int)

{

TIterator<T> iter(\*this);

++(\*this);

return iter;

}

TIterator<T> operator--(int)

{

TIterator<T> iter(\*this);

--(\*this);

return iter;

}

bool operator==(TIterator<T> const &i)

{

return node == i.node;

}

bool operator!=(TIterator<T> const &i)

{

return !(\*this == i);

}

private:

std::shared\_ptr< Item<T> > node; // node или itemt?

};

#endif // TITERATOR\_H

CMakeLists.txt:

cmake\_minimum\_required(VERSION 3.10)

project(lab5)

set(CMAKE\_CXX\_STANDARD 11)

add\_executable(lab5 point.h

point.cpp

main.cpp

figure.h

triangle.h triangle.cpp

item.h item.cpp titerator.h tlinkedlist.h tlinkedlist.cpp templates.cpp)