Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Отчёт по лабораторным работам**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

**Вариант: 30**

**Контейнер 1 уровня** – Очередь.

**Контейнер 2 уровня** – Стек.

**Фигуры:** Пятиугольник/Шестиугольник/Восьмиугольник

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
|  |  |
| **Баллы:** |  |
| **Дата:** |  |

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №1**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

**Цель работы**

Целью лабораторной работы является:

• Программирование классов на языке С++

• Управление памятью в языке С++

• Изучение базовых понятий ООП.

• Знакомство с классами в C++.

• Знакомство с перегрузкой операторов.

• Знакомство с дружественными функциями.

• Знакомство с операциями ввода-вывода из стандартных библиотек.

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов задания.

Классы должны удовлетворять следующим правилам:

• Должны иметь общий родительский класс Figure.

• Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в стандартный поток вывода cout.

• Должный иметь общий виртуальный метод расчета площади фигуры – Square.

• Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного потока cin.

• Должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Программа должна позволять вводить фигуру каждого типа с клавиатуры, выводить параметры фигур на экран и их площадь.

**Код**

main.cpp

#include"Octagon.h"

#include<iostream>

int main()

{

Figure \*ptr;

std::cout << "Select figure" << std::endl;

std::cout << "1 - Pentagon" << std::endl;

std::cout << "2 - Hexagon" << std::endl;

std::cout << "3 - Octagon" << std::endl;

int n;

std::cin >> n;

switch (n)

{

case 1:

ptr = new Pentagon(std::cin);

ptr->Print();

std::cout << ptr->Square() << std::endl;

delete ptr;

break;

case 2:

ptr = new Hexagon(std::cin);

ptr->Print();

std::cout << ptr->Square() << std::endl;

delete ptr;

break;

case 3:

ptr = new Octagon(std::cin);

ptr->Print();

std::cout << ptr->Square() << std::endl;

delete ptr;

break;

}

system("pause");

return 0;

}

Figure.h

class Figure

{

public:

virtual double Square() = 0; // Площадь (?=0)

virtual void Print() = 0; // Вывод параметров фигуры и её тип

virtual ~Figure() {};

};

Pentagon.h

#include<iostream> // istream

#include"Figure.h"

class Pentagon :public Figure {

public:

Pentagon(); // Конструктор

Pentagon(std::istream &is);

Pentagon(size\_t a);

Pentagon(const Pentagon& orig); // ?Передача объекта(его параметров и типа)

double Square() override; // override - ф-ия должна переопределять вирт-ую ф-ию,

void Print() override; // чтобы не создать новую ф-ию

virtual ~Pentagon(); // Деструктор

private:

size\_t side\_a;

};

Hexagon.h

#include<iostream> // istream

#include"Pentagon.h"

class Hexagon :public Figure {

public:

Hexagon(); // Конструктор

Hexagon(std::istream &is);

Hexagon(size\_t a);

Hexagon(const Hexagon& orig); // ?Передача объекта(его параметров и типа)

double Square() override; // override - ф-ия должна переопределять вирт-ую ф-ию,

void Print() override; // чтобы не создать новую ф-ию

virtual ~Hexagon(); // Деструктор

private:

size\_t side\_a;

};

Octagon.h

#include<iostream>

#include"Hexagon.h"

class Octagon : public Figure{

public:

Octagon(); // Конструктор

Octagon(std::istream &is);

Octagon(size\_t a);

Octagon(const Octagon& orig); // ?Передача объекта(его параметров и типа)

double Square() override; // override - ф-ия должна переопределять вирт-ую ф-ию,

void Print() override; // чтобы не создать новую ф-ию

virtual ~Octagon(); // Деструктор

private:

size\_t side\_a;

};

Pentagon.cpp

#include"Pentagon.h"

#include<iostream>

#include<cmath>

Pentagon::Pentagon() : Pentagon(0) {

}

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

std::cout << "Enter side length" << std::endl;

is >> side\_a;

std::cout << "Pentagon created" << std::endl;

}

Pentagon::Pentagon(size\_t a) : side\_a(a){

std::cout << "Pentagon created: " << side\_a;

}

Pentagon::Pentagon(const Pentagon& orig) {

std::cout << "Pentagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Pentagon::Square() {

return (((double)side\_a\*(double)side\_a\*sqrt(25 + 10 \* sqrt(5))) / 4);

}

void Pentagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

Pentagon::~Pentagon() {

std::cout << "Pentagon deleted" << std::endl;

}

Hexagon.cpp

#include"Hexagon.h"

#include<iostream>

#include<cmath>

Hexagon::Hexagon() : Hexagon(0) {

}

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

std::cout << "Enter side length" << std::endl;

is >> side\_a;

std::cout << "Hexagon created" << std::endl;

}

Hexagon::Hexagon(size\_t a) : side\_a(a) {

std::cout << "Hexagon created: " << side\_a;

}

Hexagon::Hexagon(const Hexagon& orig) {

std::cout << "Hexagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Hexagon::Square() {

return (sqrt(3)\*(double)(side\_a\*side\_a)/4);

}

void Hexagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

Hexagon::~Hexagon() {

std::cout << "Hexagon deleted" << std::endl;

}

Octagon.cpp

#include"Octagon.h"

#include<iostream>

#include<cmath>

Octagon::Octagon() : Octagon(0) {

}

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

std::cout << "Enter side length" << std::endl;

is >> side\_a;

std::cout << "Octagon created" << std::endl;

}

Octagon::Octagon(size\_t a) : side\_a(a) {

std::cout << "Octagon created: " << side\_a;

}

Octagon::Octagon(const Octagon& orig) {

std::cout << "Octagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Octagon::Square() {

return (2\*(double)(side\_a\*side\_a)\*(1+sqrt(2)));

}

void Octagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

Octagon::~Octagon() {

std::cout << "Hexagon deleted" << std::endl;

}

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
|  |  |
| **Баллы:** | 10 |
| **Дата:** |  |

**Цель работы**

Целью лабораторной работы является:

• Закрепление навыков работы с классами.

• Создание простых динамических структур данных.

• Работа с объектами, передаваемыми «по значению».

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру ( колонка фигура 1), согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.

• Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<). Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).

• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>). Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).

• Классы фигур должны иметь операторы копирования (=).

• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).

• Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).

• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.

• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

• Стандартные контейнеры std.

• Шаблоны (template).

• Различные варианты умных указателей (shared\_ptr, weak\_ptr).

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.

**Код**

main.cpp

#include "TQueue.h"

#include<iostream>

int main()

{

int a = 0;

TQueue queue;

std::cout << "Menu" << std::endl;

std::cout << "1 - create pentagon" << std::endl;

std::cout << "2 - push item to queue" << std::endl;

std::cout << "3 - check queue emptiness" << std::endl;

std::cout << "4 - delete item from queue" << std::endl;

std::cout << "5 - display queue" << std::endl;

std::cout << "6 - exit" << std::endl;

std::cout << std::endl;

while (std::cin>>a)

switch (a) {

case 1: // Создать пятиугольник

{

Pentagon \*ptr = new Pentagon(std::cin);

std::cout << \*ptr;

delete ptr;

break;

}

case 2: // Добавить элемент в стек

{

queue.push(Pentagon(std::cin));

break;

}

case 3: // Не пуст ли стек

if (queue.empty()) std::cout << "Queue is empty" << std::endl;

else std::cout << "Queue is not empty" << std::endl;

break;

case 4: // Удалить элемент из стека

queue.pop();

break;

case 5: // Вывести на экран все элементы из стека

std::cout << queue;

break;

case 6:

system("pause");

return 0;

break;

}

system("pause");

return 0;

}

Pentagon.h

#include<iostream>

#include<cstdlib>

class Pentagon {

public:

Pentagon(); // Конструктор

Pentagon(std::istream &is);

Pentagon(size\_t a);

Pentagon(const Pentagon& orig); // ?Передача объекта(его параметров и типа)

double Square(); // override - ф-ия должна переопределять вирт-ую ф-ию,

void Print(); // чтобы не создать новую ф-ию

Pentagon& operator++();

Pentagon& operator--();

Pentagon& operator++(int);

Pentagon& operator--(int);

friend Pentagon operator+(const Pentagon& left, const Pentagon& right);

Pentagon operator-(const Pentagon& right);

friend std::ostream& operator<<(std::ostream& os, const Pentagon& obj);

friend std::istream& operator>>(std::istream& is, Pentagon& obj);

Pentagon& operator=(const Pentagon& right);

bool operator==(const Pentagon& right);

bool operator!=(const Pentagon& right);

virtual ~Pentagon(); // Деструктор

private:

size\_t side\_a;

};

TQueue.h

#include "TQueueItem.h"

class TQueue {

public:

TQueue();

TQueue(const TQueue& orig);

void push(Pentagon &&pentagon);

bool empty();

Pentagon pop();

friend std::ostream& operator<<(std::ostream& os, const TQueue& queue);

virtual ~TQueue();

private:

TQueueItem \*head;

};

TQueueItem.h

#include "TQueueItem.h"

class TQueue {

public:

TQueue();

TQueue(const TQueue& orig);

void push(Pentagon &&pentagon);

bool empty();

Pentagon pop();

friend std::ostream& operator<<(std::ostream& os, const TQueue& queue);

virtual ~TQueue();

private:

TQueueItem \*head;

};

Pentagon.cpp

#include"Pentagon.h"

#include<iostream>

#include<cmath>

Pentagon::Pentagon() : Pentagon(0) {

}

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

std::cout << "Enter the number" << std::endl;

is >> side\_a;

}

Pentagon::Pentagon(size\_t a) : side\_a(a){

std::cout << "Pentagon created: " << side\_a << std::endl;

}

Pentagon::Pentagon(const Pentagon& orig) {

std::cout << "Pentagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Pentagon::Square() {

return (((double)side\_a\*(double)side\_a\*sqrt(25 + 10 \* sqrt(5))) / 4);

}

void Pentagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

Pentagon::~Pentagon() {

std::cout << "Pentagon deleted" << std::endl;

}

Pentagon& Pentagon::operator=(const Pentagon& right) {

if (this == &right) return \*this;

std::cout << "Pentagon copied" << std::endl;

side\_a = right.side\_a;

return \*this;

}

bool Pentagon::operator==(const Pentagon& right) {

if (this->side\_a == right.side\_a) return true;

else return false;

}

bool Pentagon::operator!=(const Pentagon& right) {

if (this->side\_a != right.side\_a) return true;

else return false;

}

Pentagon& Pentagon::operator++() {

side\_a++;

return \*this;

}

Pentagon& Pentagon::operator++(int) {

side\_a++;

return \*this;

}

Pentagon& Pentagon::operator--() {

if (side\_a - 1 > 0) this->side\_a--;

return \*this;

}

Pentagon& Pentagon::operator--(int) {

if (side\_a - 1 > 0) this->side\_a--;

return \*this;

}

Pentagon operator+(const Pentagon& left,const Pentagon& right) {

return Pentagon(left.side\_a+right.side\_a);

}

Pentagon Pentagon::operator-(const Pentagon& right) {

if (this->side\_a - right.side\_a > 0) this->side\_a -= right.side\_a;

return \*this;

}

std::ostream& operator<<(std::ostream& os, const Pentagon& obj) {

os << "Figure: Pentagon" << std::endl << "a=" << obj.side\_a << std::endl;

return os;

}

std::istream& operator>>(std::istream& is, Pentagon& obj) {

is >> obj.side\_a;

return is;

}

TQueue.cpp

#include "TQueue.h"

TQueue::TQueue() : head(nullptr) {

}

TQueue::TQueue(const TQueue& orig) {

head = orig.head;

}

std::ostream& operator<<(std::ostream& os, const TQueue& queue) {

TQueueItem \*item = queue.head;

while (item != nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

void TQueue::push(Pentagon &&pentagon) {

TQueueItem \*other = new TQueueItem(pentagon);

std::cout << (head == NULL) << std::endl;

other->SetNext(head);

head = other;

}

bool TQueue::empty() {

return head == nullptr;

}

Pentagon TQueue::pop() {

/\*Pentagon result;

if (head != nullptr) {

TQueueItem \*old\_head = head;

head = head->GetNext();

result = old\_head->GetPentagon();

old\_head->SetNext(nullptr);

delete old\_head;

}

return result;\*/

Pentagon result;

if (head != nullptr) {

if (head->GetNext() == nullptr) {

result = head->GetPentagon();

delete head;

head = nullptr;

}

else {

TQueueItem \*prev=head, \*cur=head->GetNext();

while (cur->GetNext() != nullptr) {

cur = cur->GetNext();

prev = prev->GetNext();

}

prev->SetNext(nullptr);

result = cur->GetPentagon();

delete cur;

}

}

return result;

}

TQueue::~TQueue() {

delete head;

}

TQueueItem.cpp

#include"TQueueItem.h"

#include<iostream>

TQueueItem::TQueueItem(const Pentagon& pentagon) {

this->pentagon = pentagon;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

TQueueItem::TQueueItem(const TQueueItem& orig) {

this->pentagon = orig.pentagon;

this->next = orig.next;

std::cout << "Queue item: copied" << std::endl;

}

TQueueItem\* TQueueItem::SetNext(TQueueItem\* next) {

TQueueItem\* old = this->next;

this->next = next;

return old;

}

Pentagon TQueueItem::GetPentagon() const {

return this->pentagon;

}

TQueueItem\* TQueueItem::GetNext() {

return this->next;

}

TQueueItem::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

delete next;

}

std::ostream& operator<<(std::ostream& os, const TQueueItem& obj) {

os << "[" << obj.pentagon << "]" << std::endl;

return os;

}

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
|  |  |
| **Баллы:** | 10 |
| **Дата:** |  |

**Цель работы**

Целью лабораторной работы является:

• Закрепление навыков работы с классами.

• Знакомство с умными указателями.

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.

• Класс-контейнер должен соджержать объекты используя std:shared\_ptr<…>.

• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.

• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

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.

**Код**

main.cpp

#include<iostream>

#include"TQueue.h"

int main()

{

std::shared\_ptr<Figure> t;

TQueue queue;

queue.push(std::shared\_ptr<Pentagon>(new Pentagon(1)));

queue.push(std::shared\_ptr<Hexagon>(new Hexagon(0)));

queue.push(std::shared\_ptr<Octagon>(new Octagon(3)));

std::cout << queue;

t = queue.pop(); t->Print();

/\*t = stack.pop(); t->Print();

t = stack.pop(); t->Print();

t = stack.pop(); t->Print();\*/

system("pause");

return 0;

}

Figure.h

class Figure

{

public:

virtual double Square() = 0; // Площадь (?=0)

virtual void Print() = 0; // Вывод параметров фигуры и её тип

//virtual std::ostream& operator<<(const Figure& in);

virtual ~Figure() {};

};

Pentagon.h

#include<iostream>

#include"Figure.h"

class Pentagon :public Figure {

public:

Pentagon();

Pentagon(const Pentagon& orig);

Pentagon(std::istream &is);

Pentagon(size\_t a);

//friend std::istream& operator>>(std::istream& is, Pentagon& in);

friend std::ostream& operator<<(std::ostream& os, const Pentagon& in);

double Square(); // Площадь

void Print(); // Распечатать характеристики

virtual ~Pentagon(); // Деструктор

private:

size\_t side\_a=0; // Сторона правильного пятиугольника

};

Hexagon.h

#include<iostream> // istream

#include"Octagon.h"

class Hexagon :public Figure{

public:

Hexagon(); // Конструктор

Hexagon(std::istream &is);

Hexagon(size\_t a);

Hexagon(const Hexagon& orig); // ?Передача объекта(его параметров и типа)

friend std::ostream& operator<<(std::ostream& os, const Hexagon& in);

double Square() override; // override - ф-ия должна переопределять вирт-ую ф-ию,

void Print() override; // чтобы не создать новую ф-ию

virtual ~Hexagon(); // Деструктор

private:

size\_t side\_a;

};

Octagon.h

#include<iostream>

#include"Pentagon.h"

class Octagon :public Figure {

public:

Octagon(); // Конструктор

Octagon(std::istream &is);

Octagon(size\_t a);

Octagon(const Octagon& orig); // ?Передача объекта(его параметров и типа)

friend std::ostream& operator<<(std::ostream& os, const Octagon& in);

double Square(); // override - ф-ия должна переопределять вирт-ую ф-ию,

void Print(); // чтобы не создать новую ф-ию

virtual ~Octagon(); // Деструктор

private:

size\_t side\_a;

};

TQueue.h

#include "TQueueItem.h"

#include <memory>

class TQueue {

public:

TQueue();

TQueue(const TQueue& orig);

void push(std::shared\_ptr<Figure> &&figure);

bool empty();

std::shared\_ptr<Figure> pop();

friend std::ostream& operator<<(std::ostream& os, const TQueue& queue);

virtual ~TQueue();

private:

std::shared\_ptr<TQueueItem> head;

};

TQueueItem.h

#include "Hexagon.h"

#include <memory>

class TQueueItem {

public:

TQueueItem(const Pentagon& pentagon);

TQueueItem(const TQueueItem& orig);

TQueueItem(const std::shared\_ptr<Figure> &figure);

friend std::ostream& operator<<(std::ostream& os, const TQueueItem& obj);

std::shared\_ptr<TQueueItem> SetNext(std::shared\_ptr<TQueueItem> next);

std::shared\_ptr<TQueueItem> GetNext();

std::shared\_ptr<Figure> GetFigure() const;

virtual ~TQueueItem();

private:

std::shared\_ptr<Figure> figure;

std::shared\_ptr<TQueueItem> next;

};

Pentagon.cpp

#include"Pentagon.h"

Pentagon::Pentagon() : Pentagon(0) {

}

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

is >> side\_a;

}

Pentagon::Pentagon(size\_t a) : side\_a(a) {

std::cout << "Pentagon created: " << side\_a << std::endl;

}

Pentagon::Pentagon(const Pentagon& orig) {

std::cout << "Pentagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Pentagon::Square() {

return (((double)side\_a\*(double)side\_a\*sqrt(25 + 10 \* sqrt(5))) / 4);

}

void Pentagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

std::ostream& operator<<(std::ostream& os, const Pentagon& in)

{

os << "a=" << in.side\_a;

return os;

}

Pentagon::~Pentagon() {

std::cout << "Pentagon deleted" << std::endl;

}

Hexagon.cpp

#include"Hexagon.h"

#include<iostream>

#include<cmath>

Hexagon::Hexagon() : Hexagon(0) {

}

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

is >> side\_a;

}

Hexagon::Hexagon(size\_t a) : side\_a(a) {

std::cout << "Hexagon created: " << side\_a<< std::endl;

}

Hexagon::Hexagon(const Hexagon& orig) {

std::cout << "Hexagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Hexagon::Square() {

return (sqrt(3)\*(double)(side\_a\*side\_a)/4);

}

void Hexagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

std::ostream& operator<<(std::ostream& os, const Hexagon& in)

{

os << "a=" << in.side\_a;

return os;

}

Hexagon::~Hexagon() {

std::cout << "Hexagon deleted" << std::endl;

}

Octagon.cpp

#include"Octagon.h"

#include<iostream>

#include<cmath>

Octagon::Octagon() : Octagon(0) {

}

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

is >> side\_a;

}

Octagon::Octagon(size\_t a) : side\_a(a) {

std::cout << "Octagon created: " << side\_a;

}

Octagon::Octagon(const Octagon& orig) {

std::cout << "Octagon copy created" << std::endl;

side\_a = orig.side\_a;

}

double Octagon::Square() {

return (2\*(double)(side\_a\*side\_a)\*(1+sqrt(2)));

}

void Octagon::Print() {

std::cout << "a=" << side\_a << std::endl;

}

std::ostream& operator<<(std::ostream& os, const Octagon& in)

{

os << "a=" << in.side\_a;

return os;

}

Octagon::~Octagon() {

std::cout << "Hexagon deleted" << std::endl;

}

TQueue.cpp

#include "TQueue.h"

TQueue::TQueue() : head(nullptr) {

}

TQueue::TQueue(const TQueue& orig) {

head = orig.head;

}

std::ostream& operator<<(std::ostream& os, const TQueue& queue) {

std::shared\_ptr<TQueueItem> item = queue.head;

while (item != nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

void TQueue::push(std::shared\_ptr<Figure> &&figure) {

std::shared\_ptr<TQueueItem> other(new TQueueItem(figure));

other->SetNext(head);

head = other;

}

bool TQueue::empty() {

return head == nullptr;

}

std::shared\_ptr<Figure> TQueue::pop() {

std::shared\_ptr<Figure> result;

if (head != nullptr) {

if (head->GetNext() == nullptr) {

result = head->GetFigure();

head.~shared\_ptr();

head = nullptr;

}

else {

std::shared\_ptr<TQueueItem> prev=head, cur=head->GetNext();

while (cur->GetNext() != nullptr) {

cur = cur->GetNext();

prev = prev->GetNext();

}

prev->SetNext(nullptr);

result = cur->GetFigure();

cur.~shared\_ptr();

cur = nullptr;

}

}

return result;

}

TQueue::~TQueue() {

head.~shared\_ptr();

}

TQueueItem.cpp

#include"TQueueItem.h"

#include<iostream>

TQueueItem::TQueueItem(const Pentagon& pentagon) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

TQueueItem::TQueueItem(const TQueueItem& orig) {

this->figure = orig.figure;

this->next = orig.next;

std::cout << "Queue item: copied" << std::endl;

}

TQueueItem::TQueueItem(const std::shared\_ptr<Figure> &figure) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

std::shared\_ptr<TQueueItem> TQueueItem::SetNext(std::shared\_ptr<TQueueItem> next) {

std::shared\_ptr<TQueueItem> old = this->next;

this->next = next;

return old;

}

std::shared\_ptr<Figure> TQueueItem::GetFigure() const {

return this->figure;

}

std::shared\_ptr<TQueueItem> TQueueItem::GetNext() {

return this->next;

}

TQueueItem::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

next.~shared\_ptr();

}

std::ostream& operator<<(std::ostream& os, const TQueueItem& obj) {

//os << "[" << obj.figure->Print() << "]" << std::endl;

obj.figure->Print();

return os;

}

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №4**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

**Цель работы**

Целью лабораторной работы является:

• Знакомство с шаблонами классов.

• Построение шаблонов динамических структур данных.

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ шаблон класса-контейнера первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.

• Шаблон класса-контейнера должен соджержать объекты используя std:shared\_ptr<…>.

• Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.

• Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).

• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).

• Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

• Стандартные контейнеры std.

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.

**Код (основные изменения)**

main.cpp

#include <cstdlib>

#include <iostream>

#include <memory>

#include "TQueue.h"

int main(int argc, char\*\* argv) {

TQueue<Figure> figure;

std::shared\_ptr<Figure> t;

std::cout << "1 - Push pentagon" << std::endl;

std::cout << "2 - Push octagon" << std::endl;

std::cout << "3 - Push hexagon" << std::endl;

std::cout << "4 - delete item from figure" << std::endl;

std::cout << "5 - display figure" << std::endl;

for (;;) {

int n;

int tmp;

std::cin >> n;

switch (n)

{

case 1:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

figure.push(std::shared\_ptr<Figure>(new Pentagon(tmp)));

break;

case 2:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

figure.push(std::shared\_ptr<Figure>(new Octagon(tmp)));

break;

case 3:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

figure.push(std::shared\_ptr<Figure>(new Hexagon(tmp)));

break;

case 4:

if (!figure.empty()) { t = figure.pop(); t->Print(); }

else std::cout << "Stack is empty" << std::endl;

break;

case 5:

if (!figure.empty()) std::cout << figure;

else std::cout << "Stack is empty" << std::endl;

}

}

system("pause");

return 0;

}

TQueue.h

#include "TQueueItem.h"

#include <memory>

template <class T> class TQueue {

public:

TQueue();

TQueue(const TQueue& orig);

void push(std::shared\_ptr<T> &&figure);

bool empty();

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

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A>& queue);

virtual ~TQueue();

private:

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

};

TQueueItem.h

#include "Hexagon.h"

#include <memory>

template <class T> class TQueueItem {

public:

TQueueItem(const Pentagon& pentagon);

TQueueItem(const TQueueItem& orig);

TQueueItem(const std::shared\_ptr<T> &figure);

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj);

std::shared\_ptr<TQueueItem<T>> SetNext(std::shared\_ptr<TQueueItem> next);

std::shared\_ptr<TQueueItem<T>> GetNext();

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

virtual ~TQueueItem();

private:

std::shared\_ptr<T> figure;

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

};

TQueue.cpp

#include "TQueue.h"

template <class T> TQueue<T>::TQueue() : head(nullptr) {

}

template <class T> TQueue<T>::TQueue(const TQueue<T>& orig) {

head = orig.head;

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {

std::shared\_ptr<TQueueItem<T>> item = queue.head;

while (item != nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

template <class T> void TQueue<T>::push(std::shared\_ptr<T> &&figure) {

std::shared\_ptr<TQueueItem<T>> other(new TQueueItem<T>(figure));

other->SetNext(head);

head = other;

}

template <class T> bool TQueue<T>::empty() {

return head == nullptr;

}

template <class T> std::shared\_ptr<T> TQueue<T>::pop() {

std::shared\_ptr<T> result;

if (head != nullptr) {

if (head->GetNext() == nullptr) {

result = head->GetFigure();

head.~shared\_ptr();

head = nullptr;

}

else {

std::shared\_ptr<TQueueItem<T>> prev=head, cur=head->GetNext();

while (cur->GetNext() != nullptr) {

cur = cur->GetNext();

prev = prev->GetNext();

}

prev->SetNext(nullptr);

result = cur->GetFigure();

cur.~shared\_ptr();

cur = nullptr;

}

}

return result;

}

template <class T> TQueue<T>::~TQueue() {

head.~shared\_ptr();

}

template class TQueue<Figure>;

template std::ostream& operator<<(std::ostream& os, const TQueue<Figure>& stack);

TQueueItem.cpp

#include"TQueueItem.h"

#include<iostream>

template <class T> TQueueItem<T>::TQueueItem(const Pentagon& pentagon) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> TQueueItem<T>::TQueueItem(const TQueueItem<T>& orig) {

this->figure = orig.figure;

this->next = orig.next;

std::cout << "Queue item: copied" << std::endl;

}

template <class T> TQueueItem<T>::TQueueItem(const std::shared\_ptr<T> &figure) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::SetNext(std::shared\_ptr<TQueueItem<T>> next) {

std::shared\_ptr<TQueueItem> old = this->next;

this->next = next;

return old;

}

template <class T> std::shared\_ptr<T> TQueueItem<T>::GetFigure() const {

return this->figure;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::GetNext() {

return this->next;

}

template <class T> TQueueItem<T>::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

next.~shared\_ptr();

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueueItem<T>& obj) {

//os << "[" << obj.figure->Print() << "]" << std::endl;

obj.figure->Print();

return os;

}

//#include "Figure.h"

template class TQueueItem<Figure>;

template std::ostream& operator<<(std::ostream& os, const TQueueItem<Figure>& obj);

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

**Введение**

В данной лабораторной работе я познакомлюсь с итераторами и научусь их применять на практике. Они облегчат проход по динамическим структурам данных.

**Цель работы**

Целью лабораторной работы является:

• Закрепление навыков работы с шаблонами классов.

• Построение итераторов для динамических структур данных.

**Задание**

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№4) спроектировать и разработать Итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми 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.

**Код (основные изменения)**

main.cpp

#include <cstdlib>

#include <iostream>

#include <memory>

#include "TQueue.h"

// template stack on shared\_ptr

int main(int argc, char\*\* argv) {

TQueue<Figure> queue;

std::shared\_ptr<Figure> t;

std::cout << "1 - Push pentagon" << std::endl;

std::cout << "2 - Push octagon" << std::endl;

std::cout << "3 - Push hexagon" << std::endl;

std::cout << "4 - delete item from stack" << std::endl;

std::cout << "5 - display stack" << std::endl;

std::cout << "6 - display stack (for)" << std::endl;

std::cout << "7 - exit" << std::endl;

for (;;) {

int n;

int tmp;

std::cin >> n;

switch (n)

{

case 1:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Pentagon(tmp)));

break;

case 2:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Octagon(tmp)));

break;

case 3:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Hexagon(tmp)));

break;

case 4:

if (!queue.empty()) { t = queue.pop(); t->Print(); }

else std::cout << "Stack is empty" << std::endl;

break;

case 5:

if (!queue.empty()) std::cout << queue;

else std::cout << "Stack is empty" << std::endl;

break;

case 6:

for (auto i : queue) i->Print();

break;

case 7:

system("pause");

return 0;

}

}

system("pause");

return 0;

}

TQueue.h

#include "TQueueItem.h"

#include "TIterator.h"

#include <memory>

template <class T> class TQueue {

public:

TQueue();

TQueue(const TQueue& orig);

void push(std::shared\_ptr<T> &&figure);

bool empty();

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

TIterator<TQueueItem<T>, T> begin();

TIterator<TQueueItem<T>, T> end();

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A>& queue);

virtual ~TQueue();

private:

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

};

TQueueItem.h

#include "Hexagon.h"

#include <memory>

template <class T> class TQueueItem {

public:

TQueueItem(const Pentagon& pentagon);

TQueueItem(const TQueueItem& orig);

TQueueItem(const std::shared\_ptr<T> &figure);

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj);

std::shared\_ptr<TQueueItem<T>> SetNext(std::shared\_ptr<TQueueItem> next);

std::shared\_ptr<TQueueItem<T>> GetNext();

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

void \* operator new (size\_t size);

void operator delete(void \*p);

virtual ~TQueueItem();

private:

std::shared\_ptr<T> figure;

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

};

TQueue.cpp

#include "TQueue.h"

template <class T> TQueue<T>::TQueue() : head(nullptr) {

}

template <class T> TQueue<T>::TQueue(const TQueue<T>& orig) {

head = orig.head;

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {

std::shared\_ptr<TQueueItem<T>> item = queue.head;

while (item != nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

template <class T> void TQueue<T>::push(std::shared\_ptr<T> &&figure) {

std::shared\_ptr<TQueueItem<T>> other(new TQueueItem<T>(figure));

other->SetNext(head);

head = other;

}

template <class T> bool TQueue<T>::empty() {

return head == nullptr;

}

template <class T> std::shared\_ptr<T> TQueue<T>::pop() {

std::shared\_ptr<T> result;

if (head != nullptr) {

if (head->GetNext() == nullptr) {

result = head->GetFigure();

head.~shared\_ptr();

head = nullptr;

}

else {

std::shared\_ptr<TQueueItem<T>> prev=head, cur=head->GetNext();

while (cur->GetNext() != nullptr) {

cur = cur->GetNext();

prev = prev->GetNext();

}

prev->SetNext(nullptr);

result = cur->GetFigure();

cur.~shared\_ptr();

cur = nullptr;

}

}

return result;

}

template <class T> TIterator<TQueueItem<T>, T> TQueue<T>::begin()

{

return TIterator<TQueueItem<T>, T>(head);

}

template <class T> TIterator<TQueueItem<T>, T> TQueue<T>::end()

{

return TIterator<TQueueItem<T>, T>(nullptr);

}

template <class T> TQueue<T>::~TQueue() {

head.~shared\_ptr();

}

template class TQueue<Figure>;

template std::ostream& operator<<(std::ostream& os, const TQueue<Figure>& queue);

TQueueItem.cpp

#include"TQueueItem.h"

#include<iostream>

template <class T> TQueueItem<T>::TQueueItem(const Pentagon& pentagon) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> TQueueItem<T>::TQueueItem(const TQueueItem<T>& orig) {

this->figure = orig.figure;

this->next = orig.next;

std::cout << "Queue item: copied" << std::endl;

}

template <class T> TQueueItem<T>::TQueueItem(const std::shared\_ptr<T> &figure) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::SetNext(std::shared\_ptr<TQueueItem<T>> next) {

std::shared\_ptr<TQueueItem> old = this->next;

this->next = next;

return old;

}

template <class T> std::shared\_ptr<T> TQueueItem<T>::GetFigure() const {

return this->figure;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::GetNext() {

return this->next;

}

template <class T> TQueueItem<T>::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

next.~shared\_ptr();

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueueItem<T>& obj) {

//os << "[" << obj.figure->Print() << "]" << std::endl;

obj.figure->Print();

return os;

}

template <class T> void \* TQueueItem<T>::operator new (size\_t size) {

std::cout << "Allocated :" << size << "bytes" << std::endl;

return malloc(size);

}

template <class T> void TQueueItem<T>::operator delete(void \*p) {

std::cout << "Deleted" << std::endl;

free(p);

}

//#include "Figure.h"

template class TQueueItem<Figure>;

template std::ostream& operator<<(std::ostream& os, const TQueueItem<Figure>& obj);

TIterator.h

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <memory>

#include <iostream>

template <class node, class T>

class TIterator

{

public:

TIterator(std::shared\_ptr<node> n) {

node\_ptr = n;

}

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

return node\_ptr->GetFigure();

}

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

return node\_ptr->GetFigure();

}

void operator ++ (){

node\_ptr = node\_ptr->GetNext();

}

TIterator operator ++ (int){

TIterator iter(\*this);

++(\*this);

return iter;

}

bool operator == (TIterator const& i){

return node\_ptr == i.node\_ptr;

}

bool operator != (TIterator const& i){

return !(\*this == i);

}

private:

std::shared\_ptr<node> node\_ptr;

};

#endif /\* TITERATOR\_H \*/

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

**Цель работы**

Целью лабораторной работы является:

• Закрепление навыков по работе с памятью в C++.

• Создание аллокаторов памяти для динамических структур данных.

**Задание**

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№5) спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианта задания).

Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.

Нельзя использовать:

• Стандартные контейнеры std.

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.• Удалять фигуры из контейнера.

**Код (основные изменения)**

TIteratorStack.h

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <memory>

#include <iostream>

#include "TStackItem.h"

class TIteratorStack

{

public:

TIteratorStack(TStackItem \*n) {

node\_ptr = n;

}

void \*operator \* (){

return node\_ptr->GetBlock();

}

void \*operator -> (){

return node\_ptr->GetBlock();

}

void operator ++ (){

node\_ptr = node\_ptr->GetNext();

}

TIteratorStack operator ++ (int){

TIteratorStack iter(\*this);

++(\*this);

return iter;

}

bool operator == (TIteratorStack const& i){

return node\_ptr == i.node\_ptr;

}

bool operator != (TIteratorStack const& i){

return !(\*this == i);

}

TIteratorStack operator=(void \*ptr) {

\*this->node\_ptr = ptr;

return \*this;

}

private:

TStackItem \*node\_ptr;

};

#endif /\* TITERATOR\_H \*/

TQueueItem.h

#include "Hexagon.h"

#include "TAllocationBlock.h"

#include <memory>

template <class T> class TQueueItem {

public:

TQueueItem(const Pentagon& pentagon);

TQueueItem(const TQueueItem& orig);

TQueueItem(const std::shared\_ptr<T> &figure);

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj);

std::shared\_ptr<TQueueItem<T>> SetNext(std::shared\_ptr<TQueueItem> next);

std::shared\_ptr<TQueueItem<T>> GetNext();

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

void \* operator new (size\_t size);

void operator delete(void \*p);

virtual ~TQueueItem();

private:

std::shared\_ptr<T> figure;

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

static TAllocationBlock stackitem\_allocator;

};

TStack.h

#ifndef TSTACK\_H

#define TSTACK\_H

#include "TIteratorStack.h"

class TStack {

public:

TStack();

TStack(const TStack& orig);

void push(void \*block);

bool empty();

//void \*pop();

TIteratorStack begin();

TIteratorStack end();

virtual ~TStack();

private:

TStackItem \*head;

};

#endif /\* TSTACK\_H \*/

TStackItem.h

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

class TStackItem {

public:

TStackItem(void \*block);

//TStackItem(const TStackItem& orig);

TStackItem\* SetNext(TStackItem\* next);

TStackItem\* GetNext();

void \*GetBlock() const;

TStackItem \* operator=(void \*ptr);

virtual ~TStackItem();

private:

void \*block;

TStackItem \*next;

};

#endif /\* TSTACKITEM\_H \*/

TQueueItem.cpp

#include"TQueueItem.h"

#include<iostream>

template <class T> TQueueItem<T>::TQueueItem(const Pentagon& pentagon) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> TQueueItem<T>::TQueueItem(const TQueueItem<T>& orig) {

this->figure = orig.figure;

this->next = orig.next;

std::cout << "Queue item: copied" << std::endl;

}

template <class T> TQueueItem<T>::TQueueItem(const std::shared\_ptr<T> &figure) {

this->figure = figure;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> TAllocationBlock TQueueItem<T>::stackitem\_allocator(sizeof(TQueueItem<T>), 100);

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::SetNext(std::shared\_ptr<TQueueItem<T>> next) {

std::shared\_ptr<TQueueItem> old = this->next;

this->next = next;

return old;

}

template <class T> std::shared\_ptr<T> TQueueItem<T>::GetFigure() const {

return this->figure;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::GetNext() {

return this->next;

}

template <class T> TQueueItem<T>::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

next.~shared\_ptr();

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueueItem<T>& obj) {

//os << "[" << obj.figure->Print() << "]" << std::endl;

obj.figure->Print();

return os;

}

template <class T> void \* TQueueItem<T>::operator new (size\_t size) {

//std::cout << "Allocated :" << size << "bytes" << std::endl;

return stackitem\_allocator.allocate();

}

template <class T> void TQueueItem<T>::operator delete(void \*p) {

std::cout << "Deleted" << std::endl;

stackitem\_allocator.deallocate(p);

}

//#include "Figure.h"

template class TQueueItem<Figure>;

template std::ostream& operator<<(std::ostream& os, const TQueueItem<Figure>& obj);

TStack.cpp

#include "TStack.h"

#include "TIterator.h"

TStack::TStack() : head(nullptr) {

}

TStack::TStack(const TStack& orig) {

head = orig.head;

}

void TStack::push(void \*block) {

TStackItem \*other = new TStackItem(block);

other->SetNext(head);

head = other;

}

bool TStack::empty() {

return head == nullptr;

}

/\*void\* TStack::pop() {

void \*result = nullptr;

if (head != nullptr) {

TStackItem \*old\_head = head;

head = head->GetNext();

result = old\_head->GetBlock();

old\_head->SetNext(nullptr);

delete old\_head;

}

return result;

}\*/

TIteratorStack TStack::begin()

{

return TIteratorStack(head);

}

TIteratorStack TStack::end()

{

return TIteratorStack(nullptr);

}

TStack::~TStack() {

delete head;

}

TStackItem.cpp

#include "TStackItem.h"

#include <iostream>

TStackItem::TStackItem(void \*block) {

this->block = block;

this->next = nullptr;

std::cout << "Stack item: created" << std::endl;

}

/\*TStackItem::TStackItem(const TStackItem& orig) {

this->triangle = orig.triangle;

this->next = orig.next;

std::cout << "Stack item: copied" << std::endl;

}\*/

TStackItem\* TStackItem::SetNext(TStackItem\* next) {

TStackItem\* old = this->next;

this->next = next;

return old;

}

void\* TStackItem::GetBlock() const {

return this->block;

}

TStackItem \* TStackItem::operator=(void \*ptr) {

this->block = ptr;

return this;

}

TStackItem\* TStackItem::GetNext() {

return this->next;

}

TStackItem::~TStackItem() {

std::cout << "Stack item: deleted" << std::endl;

delete next;

}

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №7**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

**Цель работы**

Целью лабораторной работы является:

• Создание сложных динамических структур данных.

• Закрепление принципа OCP.

**Задание**

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер, одного из следующих видов (Контейнер 1-го уровня):

1. Массив

2. Связанный список

3. Бинарное- Дерево.

4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).

5. Очередь

6. Стек

Каждым элементом контейнера, в свою, является динамической структурой данных одного из следующих видов (Контейнер 2-го уровня):

1. Массив

2. Связанный список

3. Бинарное- Дерево

4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).

5. Очередь

6. Стек

Таким образом у нас получается контейнер в контейнере. Т.е. для варианта (1,2) это будет массив, каждый из элементов которого – связанный список. А для варианта (5,3) – это очередь из бинарных деревьев.

Элементом второго контейнера является объект-фигура, определенная вариантом задания.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше 5. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Например, для варианта (1,2) добавление объектов будет выглядеть следующим образом:

1. Вначале массив пустой.

2. Добавляем Объект1: В массиве по индексу 0 создается элемент с типом список, в список

добавляется Объект 1.

3. Добавляем Объект2: Объект добавляется в список, находящийся в массиве по индекс 0.

4. Добавляем Объект3: Объект добавляется в список, находящийся в массиве по индекс 0.

5. Добавляем Объект4: Объект добавляется в список, находящийся в массиве по индекс 0. 6. Добавляем Объект5: Объект добавляется в список, находящийся в массиве по индекс 0.

7. Добавляем Объект6: В массиве по индексу 1 создается элемент с типом список, в список добавляется Объект 6.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта (в том числе и для деревьев).

При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалится.

Нельзя использовать:

• Стандартные контейнеры std.

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера (1-го и 2-го уровня).

• Удалять фигуры из контейнера по критериям:

o По типу (например, все квадраты).

o По площади (например, все объекты с площадью меньше чем заданная).

**Код (основные изменения)**

**main.cpp**

**#include <cstdlib>**

#include <iostream>

#include <memory>

#include "TQueue.h"

#include "IRemoveCriteriaByValue.h"

// template stack on shared\_ptr

int main(int argc, char\*\* argv) {

TQueue<Figure> queue;

IRemoveCriteriaByValue<Figure> criteria(std::shared\_ptr<Figure>(new Pentagon(3)));

std::shared\_ptr<Figure> t;

std::cout << "1 - Push pentagon" << std::endl;

std::cout << "2 - Push octagon" << std::endl;

std::cout << "3 - Push hexagon" << std::endl;

std::cout << "4 - delete item from stack" << std::endl;

std::cout << "5 - display stack" << std::endl;

std::cout << "6 - display stack (for)" << std::endl;

std::cout << "7 - exit" << std::endl;

for (;;) {

int n;

int tmp;

std::cin >> n;

switch (n)

{

case 1:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Pentagon(tmp)));

break;

case 2:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Octagon(tmp)));

break;

case 3:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Hexagon(tmp)));

break;

case 4:

if (!queue.empty()) { t = queue.pop(); }//t->Print(); }

else std::cout << "Stack is empty" << std::endl;

break;

case 5:

if (!queue.empty()) std::cout << queue;

else std::cout << "Stack is empty" << std::endl;

break;

case 6:

//int numt;

//std::cin >> numt;

//IRemoveCriteriaByValue<Figure> criteria(std::shared\_ptr<Figure>(new Pentagon(numt)));

queue.RemoveSubitem(&criteria);

//for (auto i : queue) i->Print();

break;

case 7:

system("pause");

return 0;

}

}

system("pause");

return 0;

}

**TQueue.h**

**#include "TQueueItem.h"**

#include "TIterator.h"

#include "IRemoveCriteria.h"

#include <memory>

template <class T> class TQueue {

public:

TQueue();

TQueue(const TQueue& orig);

void push(std::shared\_ptr<T> &&figure);

bool empty();

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

void RemoveSubitem(IRemoveCriteria<T> \* criteria);

TIterator<TQueueItem<T>, T> begin();

TIterator<TQueueItem<T>, T> end();

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A>& queue);

virtual ~TQueue();

private:

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

**};**

**TQueueItem.h**

**#include <memory>**

#include "TStack.h"

template <class T> class TQueueItem {

public:

//TQueueItem(const Pentagon& pentagon);

//TQueueItem(const TQueueItem& orig);

TQueueItem(const std::shared\_ptr<T> &figure);

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj);

std::shared\_ptr<TQueueItem<T>> SetNext(std::shared\_ptr<TQueueItem> next);

std::shared\_ptr<TQueueItem<T>> GetNext();

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

TStack<T> stack;

int length = 0;

void \* operator new (size\_t size);

void operator delete(void \*p);

virtual ~TQueueItem();

private:

//std::shared\_ptr<T> figure;

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

**};**

**TStack.h**

**#ifndef TSTACK\_H**

#define TSTACK\_H

#include "Hexagon.h"

#include "TStackItem.h"

#include "TIterator.h"

#include <memory>

template <class T> class TStack {

public:

TStack();

void push(std::shared\_ptr<T> &item);

void push(const std::shared\_ptr<T> &item);

bool empty();

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

TIterator<TStackItem<T>, T> begin();

TIterator<TStackItem<T>, T> end();

template <class A> friend std::ostream& operator<<(std::ostream& os,const TStack<A>& stack);

virtual ~TStack();

private:

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

};

#endif /\* TSTACK\_H \*/

**TStackItem.h**

**#ifndef TSTACKITEM\_H**

#define TSTACKITEM\_H

#include <memory>

template<class T> class TStackItem {

public:

TStackItem(const std::shared\_ptr<T>& figure);

template<class A> friend std::ostream& operator<<(std::ostream& os, const TStackItem<A>& obj);

std::shared\_ptr<TStackItem<T>> SetNext(std::shared\_ptr<TStackItem> &next);

std::shared\_ptr<TStackItem<T>> GetNext();

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

void \* operator new (size\_t size);

void operator delete(void \*p);

virtual ~TStackItem();

private:

std::shared\_ptr<T> item;

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

// std::weak\_ptr<TStackItem<T>> prev;

};

#endif /\* TSTACKITEM\_H \*/

**TQueue.cpp**

**#include "TQueue.h"**

template <class T> TQueue<T>::TQueue() : head(nullptr) {

}

template <class T> TQueue<T>::TQueue(const TQueue<T>& orig) {

head = orig.head;

}

template <class T> void TQueue<T>::RemoveSubitem(IRemoveCriteria<T> \* criteria) {

std::shared\_ptr<TQueueItem<T>> cur = head;

while (cur != nullptr) {

TStack<T> stackCopy;

while (!cur->stack.empty()) {

std::shared\_ptr<T> curI = cur->stack.pop();

if (criteria->isIt(curI)) {

std::cout << "Stack: Delete element " << std::endl;

cur->length--;

}

else

stackCopy.push(curI);

}

if (stackCopy.empty()) {

std::shared\_ptr<TQueueItem<T>> start = head;

if (cur == head) {

head = head->GetNext();

}

else {

while (start->GetNext() != cur) {

start = start->GetNext();

}

start->SetNext(cur->GetNext());

//cur = nullptr;

cur = start;

}

}

/\*if (stackCopy.empty()) {

std::shared\_ptr<TQueueItem<T>> start = head;

if (start == cur) {

std::cout<<"here";

head = head->GetNext();

cur = head;

continue;

}

while (start->GetNext() != cur) {

start = start->GetNext();

}

start->SetNext(cur->GetNext());

cur = start->GetNext();

continue;

}\*/

while (!stackCopy.empty())

cur->stack.push(stackCopy.pop());

cur = cur->GetNext();

}

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {

std::shared\_ptr<TQueueItem<T>> item = queue.head;

while (item != nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

template <class T> void TQueue<T>::push(std::shared\_ptr<T> &&figure) {

std::shared\_ptr<TQueueItem<T>> cur = head;

bool ok = false;

while (cur != nullptr) {

if (cur->length < 5) {

cur->stack.push(figure);

cur->length++;

ok = true;

std::cout << cur->length;

break;

}

else {

cur = cur->GetNext();

}

}

if (!ok) {

std::shared\_ptr<TQueueItem<T>> other(new TQueueItem<T>(figure));

other->SetNext(head);

head = other;

head->length++;

}

}

template <class T> bool TQueue<T>::empty() {

return head == nullptr;

}

template <class T> std::shared\_ptr<T> TQueue<T>::pop() {

std::shared\_ptr<T> result;

if (head != nullptr) {

if (head->GetNext() == nullptr) {

//result = head->GetFigure();

head = nullptr;

}

else {

std::shared\_ptr<TQueueItem<T>> prev=head, cur=head->GetNext();

while (cur->GetNext() != nullptr) {

cur = cur->GetNext();

prev = prev->GetNext();

}

prev->SetNext(nullptr);

//result = cur->GetFigure();

cur = nullptr;

}

}

return result;

}

template <class T> TIterator<TQueueItem<T>, T> TQueue<T>::begin()

{

return TIterator<TQueueItem<T>, T>(head);

}

template <class T> TIterator<TQueueItem<T>, T> TQueue<T>::end()

{

return TIterator<TQueueItem<T>, T>(nullptr);

}

template <class T> TQueue<T>::~TQueue() {

head.~shared\_ptr();

}

template class TQueue<Figure>;

**template std::ostream& operator<<(std::ostream& os, const TQueue<Figure>& queue);**

**TQueueItem.cpp**

**#include"TQueueItem.h"**

#include<iostream>

template <class T> TQueueItem<T>::TQueueItem(const std::shared\_ptr<T> &figure) {

//this->figure = figure;

this->stack.push(figure);

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::SetNext(std::shared\_ptr<TQueueItem<T>> next) {

std::shared\_ptr<TQueueItem> old = this->next;

this->next = next;

return old;

}

template <class T> std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::GetNext() {

return this->next;

}

template <class T> TQueueItem<T>::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

next.~shared\_ptr();

}

template <class T> std::ostream& operator<<(std::ostream& os, const TQueueItem<T>& obj) {

std::cout << "["<<std::endl<<obj.stack<<"]"<<std::endl;

return os;

}

template <class T> void \* TQueueItem<T>::operator new (size\_t size) {

std::cout << "Allocated :" << size << "bytes" << std::endl;

return malloc(size);

}

template <class T> void TQueueItem<T>::operator delete(void \*p) {

std::cout << "Deleted" << std::endl;

free(p);

}

//#include "Figure.h"

template class TQueueItem<Figure>;

**template std::ostream& operator<<(std::ostream& os, const TQueueItem<Figure>& obj);**

**TStack.cpp**

**#include "TStack.h"**

template <class T> TStack<T>::TStack() : head(nullptr) {

}

template <class T> std::ostream& operator<<(std::ostream& os, const TStack<T>& stack) {

std::shared\_ptr<TStackItem<T>> item = stack.head;

while(item!=nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

template <class T> void TStack<T>::push(std::shared\_ptr<T> &item) {

if (head == nullptr) {

std::shared\_ptr<TStackItem<T>> other(new TStackItem<T>(item));

other->SetNext(head);

head = other;

}

else {

std::shared\_ptr<TStackItem<T>> other(new TStackItem<T>(item));

std::shared\_ptr<TStackItem<T>> cur = head;

while ((cur->GetNext()!=nullptr)&&(cur->GetFigure()->Square() < item->Square())) {

cur = cur->GetNext();

}

if (cur->GetNext() == nullptr) {

if (cur == head) {

if (cur->GetFigure()->Square() < item->Square()) {

head->SetNext(other);

}

else {

other->SetNext(head);

head = other;

}

return;

}

else

{

//cur->SetNext(other);

if (cur->GetFigure()->Square() < item->Square()) {

cur->SetNext(other);

}

else {

std::shared\_ptr<TStackItem<T>> curT = head;

while (curT->GetNext() != cur) {

curT = curT->GetNext();

}

curT->SetNext(other);

other->SetNext(cur);

}

}

return;

}

else {

if (cur == head) {

if (cur->GetFigure()->Square() < item->Square()) {

head->SetNext(other);

}

else {

other->SetNext(head);

head = other;

}

return;

}

else {

std::shared\_ptr<TStackItem<T>> curc = head;

while (curc->GetNext() != cur) {

curc = curc->GetNext();

}

curc->SetNext(other);

other->SetNext(cur);

}

}

}

}

template <class T> void TStack<T>::push(const std::shared\_ptr<T> &item) {

if (head == nullptr) {

std::shared\_ptr<TStackItem<T>> other(new TStackItem<T>(item));

other->SetNext(head);

head = other;

}

else {

std::shared\_ptr<TStackItem<T>> other(new TStackItem<T>(item));

std::shared\_ptr<TStackItem<T>> cur = head;

while ((cur->GetNext() != nullptr) && (cur->GetFigure()->Square() < item->Square())) {

cur = cur->GetNext();

}

if (cur->GetNext() == nullptr) {

if (cur == head) {

if (cur->GetFigure()->Square() < item->Square()) {

head->SetNext(other);

}

else {

other->SetNext(head);

head = other;

}

return;

}

else

{

//cur->SetNext(other);

if (cur->GetFigure()->Square() < item->Square()) {

cur->SetNext(other);

}

else {

std::shared\_ptr<TStackItem<T>> curT = head;

while (curT->GetNext() != cur) {

curT = curT->GetNext();

}

curT->SetNext(other);

other->SetNext(cur);

}

}

return;

}

else {

if (cur == head) {

if (cur->GetFigure()->Square() < item->Square()) {

head->SetNext(other);

}

else {

other->SetNext(head);

head = other;

}

return;

}

else {

std::shared\_ptr<TStackItem<T>> curc = head;

while (curc->GetNext() != cur) {

curc = curc->GetNext();

}

curc->SetNext(other);

other->SetNext(cur);

}

}

}

}

template <class T> bool TStack<T>::empty() {

return head == nullptr;

}

template <class T> std::shared\_ptr<T> TStack<T>::pop() {

std::shared\_ptr<T> result;

if (head != nullptr) {

result = head->GetFigure();

head = head->GetNext();

}

return result;

}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::begin()

{

return TIterator<TStackItem<T>, T>(head);

}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::end()

{

return TIterator<TStackItem<T>, T>(nullptr);

}

template <class T> TStack<T>::~TStack() {

}

//#include "Figure.h"

template class TStack<Figure>;

**template std::ostream& operator<<(std::ostream& os, const TStack<Figure>& stack);**

**TStackItem.cpp**

#include "TStackItem.h"

#include <iostream>

template <class T> TStackItem<T>::TStackItem(const std::shared\_ptr<T>& item) {

this->item = item;

this->next = nullptr;

std::cout << "Stack item: created" << std::endl;

}

template <class T> std::shared\_ptr<TStackItem<T>> TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> &next) {

std::shared\_ptr<TStackItem<T>> old = this->next;

this->next = next;

return old;

}

template <class T> std::shared\_ptr<T> TStackItem<T>::GetFigure() const {

return this->item;

}

template <class T> std::shared\_ptr<TStackItem<T>> TStackItem<T>::GetNext() {

return this->next;

}

template <class T> TStackItem<T>::~TStackItem() {

std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const TStackItem<A>& obj) {

obj.item->Print();

return os;

}

template <class T> void \* TStackItem<T>::operator new (size\_t size) {

std::cout << "Allocated :" << size << "bytes" << std::endl;

return malloc(size);

}

template <class T> void TStackItem<T>::operator delete(void \*p) {

std::cout << "Deleted" << std::endl;

free(p);

}

#include "Figure.h"

template class TStackItem<Figure>;

**template std::ostream& operator<<(std::ostream& os, const TStackItem<Figure>& obj);**

**IRemoveCriteria.h**

#ifndef IREMOVECRITERIA\_H

#define IREMOVECRITERIA\_H

template <class T> class IRemoveCriteria {

public:

virtual bool isIt(std::shared\_ptr<T> &value) = 0;

private:

};

#endif /\* IREMOVECRITERIA\_H \*/

**IRemoveCriteriaAll.h**

**#ifndef IREMOVECRITERIAALL\_H**

#define IREMOVECRITERIAALL\_H

#include "IRemoveCriteria.h"

template <class T> class IRemoveCriteriaAll : public IRemoveCriteria<T>{

public:

IRemoveCriteriaAll() {};

bool isIt(std::shared\_ptr<T> &value) override{

return true;

}

private:

};

#endif /\* IREMOVECRITERIAALL\_H \*/

**IRemoveCriteriaByValue.h**

**#ifndef IREMOVECRITERIABYVALUE\_H**

#define IREMOVECRITERIABYVALUE\_H

#include "IRemoveCriteriaAll.h"

template <class T> class IRemoveCriteriaByValue : public IRemoveCriteria<T>{

public:

IRemoveCriteriaByValue(const std::shared\_ptr<T> && value) : \_value(value) {};

bool isIt(std::shared\_ptr<T> &value) override{

return \_value->Square()==value->Square();

}

private:

std::shared\_ptr<T> \_value;

};

#endif /\* IREMOVECRITERIABYVALUE\_H \*/

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №8**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

Цель работы

Целью лабораторной работы является:

• Знакомство с параллельным программированием в C++.

Задание

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера .

Необходимо разработать два вида алгоритма:

• Обычный, без параллельных вызовов.

• С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

• future

• packaged\_task/async

Для обеспечения потоко-безопасности структур данных использовать:

• mutex

• lock\_guard

Нельзя использовать:

• Стандартные контейнеры std.

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.

• Проводить сортировку контейнера

**Код (основные изменения)**

main.cpp

#include <cstdlib>

#include <iostream>

#include <memory>

#include <functional>

#include <random>

#include <future>

#include <thread>

#include "TQueue.h"

#include "TStack.h"

// template stack on shared\_ptr

int main(int argc, char\*\* argv) {

TStack<Figure> stack\_figure;

typedef std::function<void(void)> command;

TStack <command> stack\_cmd;

command cmd\_insert = [&]() {

std::cout << "Command: Create triangles" << std::endl;

std::default\_random\_engine generator;

std::uniform\_int\_distribution<int> distribution(1, 1000);

for (int i = 0; i < 10; i++) {

int side = distribution(generator);

stack\_figure.push(std::shared\_ptr<Figure>(new Pentagon(side)));

}

};

command cmd\_print = [&]() {

std::cout << "Command: Print stack" << std::endl;

std::cout << stack\_figure;

};

command cmd\_reverse = [&]() {

std::cout << "Command: Reverse stack" << std::endl;

TStack<Figure> stack\_tmp;

while (!stack\_figure.empty()) stack\_tmp.push(stack\_figure.pop\_last());

while (!stack\_tmp.empty()) stack\_figure.push(stack\_tmp.pop());

};

stack\_cmd.push(std::shared\_ptr<command>(&cmd\_print, [](command\*) {

})); // using custom deleter

stack\_cmd.push(std::shared\_ptr<command>(&cmd\_reverse, [](command\*) {

})); // using custom deleter

stack\_cmd.push(std::shared\_ptr<command>(&cmd\_print, [](command\*) {

})); // using custom deleter

stack\_cmd.push(std::shared\_ptr<command>(&cmd\_insert, [](command\*) {

})); // using custom deleter

while (!stack\_cmd.empty()) {

std::shared\_ptr<command> cmd = stack\_cmd.pop();

std::future<void> ft = std::async(\*cmd);

ft.get();

}

system("pause");

return 0;

}

Tstack.h

#ifndef TSTACK\_H

#define TSTACK\_H

#include "Hexagon.h"

#include "TStackItem.h"

#include "TIterator.h"

#include <memory>

#include <future>

#include <mutex>

#include <thread>

template <class T> class TStack {

public:

TStack();

void push(std::shared\_ptr<T> &&item);

bool empty();

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

std::shared\_ptr<T> pop\_last();

TIterator<TStackItem<T>, T> begin();

TIterator<TStackItem<T>, T> end();

template <class A> friend std::ostream& operator<<(std::ostream& os,const TStack<A>& stack);

virtual ~TStack();

private:

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

};

#endif /\* TSTACK\_H \*/

TStackItem.h

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <memory>

#include <thread>

#include <mutex>

template<class T> class TStackItem {

public:

TStackItem(const std::shared\_ptr<T>& figure);

template<class A> friend std::ostream& operator<<(std::ostream& os, const TStackItem<A>& obj);

std::shared\_ptr<TStackItem<T>> SetNext(std::shared\_ptr<TStackItem> next);

std::shared\_ptr<TStackItem<T>> GetNext();

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

void \* operator new (size\_t size);

void operator delete(void \*p);

virtual ~TStackItem();

private:

std::shared\_ptr<T> item;

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

// std::weak\_ptr<TStackItem<T>> prev;

};

#endif /\* TSTACKITEM\_H \*/

TStack.cpp

#include "TStack.h"

#include <exception>

template <class T> TStack<T>::TStack() : head(nullptr) {

}

template <class T> std::ostream& operator<<(std::ostream& os, const TStack<T>& stack) {

std::shared\_ptr<TStackItem<T>> item = stack.head;

while(item!=nullptr)

{

os << \*item;

item = item->GetNext();

}

return os;

}

template <class T> void TStack<T>::push(std::shared\_ptr<T> &&item) {

std::shared\_ptr<TStackItem<T>> other(new TStackItem<T>(item));

other->SetNext(head);

head = other;

}

template <class T> bool TStack<T>::empty() {

return head == nullptr;

}

template <class T> std::shared\_ptr<T> TStack<T>::pop() {

std::shared\_ptr<T> result;

if (head != nullptr) {

result = head->GetFigure();

head = head->GetNext();

}

return result;

}

template <class T> std::shared\_ptr<T> TStack<T>::pop\_last() {

std::shared\_ptr<T> result;

if (head != nullptr) {

std::shared\_ptr<TStackItem < T>> element = head;

std::shared\_ptr<TStackItem < T>> prev = nullptr;

while (element->GetNext() != nullptr) {

prev = element;

element = element->GetNext();

}

if (prev != nullptr) {

prev->SetNext(nullptr);

result = element->GetFigure();

}

else {

result = element->GetFigure();

head = nullptr;

}

}

return result;

}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::begin()

{

return TIterator<TStackItem<T>, T>(head);

}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::end()

{

return TIterator<TStackItem<T>, T>(nullptr);

}

template <class T> TStack<T>::~TStack() {

}

//#include "Figure.h"

template class TStack<Figure>;

template class TStack<std::function<void(void)>>;

template std::ostream& operator<<(std::ostream& os, const TStack<Figure>& stack);

TStackItem.cpp

#include "TStackItem.h"

#include <iostream>

template <class T> TStackItem<T>::TStackItem(const std::shared\_ptr<T>& item) {

this->item = item;

this->next = nullptr;

std::cout << "Stack item: created" << std::endl;

}

template <class T> std::shared\_ptr<TStackItem<T>> TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> next) {

std::shared\_ptr<TStackItem<T>> old = this->next;

this->next = next;

return old;

}

template <class T> std::shared\_ptr<T> TStackItem<T>::GetFigure() const {

return this->item;

}

template <class T> std::shared\_ptr<TStackItem<T>> TStackItem<T>::GetNext() {

return this->next;

}

template <class T> TStackItem<T>::~TStackItem() {

std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const TStackItem<A>& obj) {

obj.item->Print();

return os;

}

template <class T> void \* TStackItem<T>::operator new (size\_t size) {

std::cout << "Allocated :" << size << "bytes" << std::endl;

return malloc(size);

}

template <class T> void TStackItem<T>::operator delete(void \*p) {

std::cout << "Deleted" << std::endl;

free(p);

}

#include "Figure.h"

template class TStackItem<Figure>;

template class TStackItem<std::function<void(void)>>;

template std::ostream& operator<<(std::ostream& os, const TStackItem<Figure>& obj);

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №9**

**по курсу “Объектно-ориентированное программирование”**

**III семестр**

|  |  |
| --- | --- |
| **Студент:** | Шарапов Л.А. |
| **Группа:** | 8О-208Б, №30 |
| **Преподаватель:** | Поповкин А.В. |
| **Вариант:** | 30 |
| **Баллы:** | 10 |
| **Дата:** |  |

Цель работы

Целью лабораторной работы является:

• Знакомство с лямбда-выражениями

Задание

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) необходимо разработать:

• Контейнер второго уровня с использованием шаблонов.

• Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контенйром 1-го уровня:

◦ Генерация фигур со случайным значением параметров;

◦ Печать контейнера на экран;

◦ Удаление элементов со значением площади меньше определенного числа;

• В контенер второго уровня поместить цепочку команд.

• Реализовать цикл, который проходит по всем командам в контенере второго уровня и выполняет их, применяя к контейнеру первого уровня.

Для создания потоков использовать механизмы:

• future

• packaged\_task/async

Для обеспечения потоко-безопасности структур данных использовать:

• mutex

• lock\_guard

Нельзя использовать:

• Стандартные контейнеры std.

**Код**

main.cpp

#include <cstdlib>

#include <iostream>

#include <memory>

#include "TQueue.h"

// template stack on shared\_ptr

int main(int argc, char\*\* argv) {

TQueue<Figure> queue;

std::shared\_ptr<Figure> t;

std::cout << "1 - Push pentagon" << std::endl;

std::cout << "2 - Push octagon" << std::endl;

std::cout << "3 - Push hexagon" << std::endl;

std::cout << "4 - delete item from stack" << std::endl;

std::cout << "5 - display stack" << std::endl;

std::cout << "6 - display stack (for)" << std::endl;

std::cout << "7 - sort" << std::endl;

std::cout << "8 - parallel sort" << std::endl;

std::cout << "9 - exit" << std::endl;

for (;;) {

int n;

int tmp;

std::cin >> n;

switch (n)

{

case 1:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Pentagon(tmp)));

break;

case 2:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Octagon(tmp)));

break;

case 3:

std::cout << "Enter number" << std::endl;

std::cin >> tmp;

queue.push(std::shared\_ptr<Figure>(new Hexagon(tmp)));

break;

case 4:

if (!queue.empty()) { t = queue.pop(); t->Print(); }

else std::cout << "Stack is empty" << std::endl;

break;

case 5:

if (!queue.empty()) std::cout << queue;

else std::cout << "Stack is empty" << std::endl;

break;

case 6:

for (auto i : queue) i->Print();

break;

case 7:

queue.sort();

break;

case 8:

queue.sort\_parallel();

break;

case 9:

system("pause");

return 0;

}

}

system("pause");

return 0;

}