**Московский авиационный институт (национальный исследовательский университет)**

**Факультет информационные технологии и прикладная математика**

**Кафедра вычислительной математики и программирования**

**Лабораторные работы по курсу «Объектно-ориентированное программирование»**

Студент: Хисамутдинов Э.А.

Преподаватель: Поповкин А.В.

Группа: 08-207

Вариант: 22

Дата:

Оценка:

Подпись:

**Москва, 2017**

**Лабораторная работа №1**

* **ЗАДАНИЕ**

Данный курс построен таким образом, что каждая новая лабораторная – это продолжение старой, поэтому в отчете приведены только основные фрагменты проекта, дабы избежать лишних повторений одинаковых программ.

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно варианту 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 расположены 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.

Фигуры: Пятиугольник, Шестиугольник, Восьмиугольник.

* **ОПИСАНИЕ**

Классы и объекты в С++ являются основными концепциями объектно-ориентированного программирования — ООП.

**Классы** в С++ — это абстракция описывающая методы, свойства, ещё не существующих объектов. **Объекты** — конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса. В ООП существует три основных принципа построения классов:

1. **Инкапсуляция** — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

2. **Наследование** — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.

3. **Полиморфизм** — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Для разграничения содержимого класса, например которое пользователю лучше не трогать, были добавлены **спецификаторы** **доступа** public, private. Это и есть инкапсуляция, которую мы упоминали выше.

• **public** — дает публичный доступ, содержимому, которое в нем указано. Так можно обратится к любой переменной или функции из любой части программы.

• **private** — запрещает обращаться к свойствам вне класса. Поэтому под крылом этого доступа часто находятся именно объявления переменных, массивов, а также прототипов функций.

**Перегрузка** **операторов** в программировании — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

**Дружественная** **функция** — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected.

**Виртуальная** **функция** — это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс.

**Конструктор** – это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями.

В отличие от конструктора, **деструктор** — специальный метод класса, который служит для уничтожения элементов класса. Чаще всего его используют тогда, когда в конструкторе при создании объекта класса динамически был выделен участок памяти и необходимо эту память очистить, если эти значения уже не нужны для дальнейшей работы программы.

**Операции ввода/вывода** выполняются с помощью классов istream (потоковый ввод) и ostream (потоковый вывод). Третий класс, iostream, является производным от них и поддерживает двунаправленный ввод/вывод. Для удобства в библиотеке определены три стандартных объекта-потока:

1. **cin** – объект класса istream, соответствующий стандартному вводу. В общем случае он позволяет читать данные с терминала пользователя;

2. **cout** – объект класса ostream, соответствующий стандартному выводу. В общем случае он позволяет выводить данные на терминал пользователя;

3. **cerr** – объект класса ostream, соответствующий стандартному выводу для ошибок. В этот поток мы направляем сообщения об ошибках программы.

* **ИСХОДНЫЙ КОД**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Pentagon.cpp | Hexagon.cpp | Octagon.cpp |
| Конструктор класса | Pentagon(); | hexagon (); | Octagon(); |
| Конструктор класса из стандартного потока | Pentagon(std::istream &is); | hexagon(std::istream &is); | Octagon(std::istream &is); |
| Конструктор копии класса | Pentagon (const Pentagon & orig); | hexagon (const hexagon & orig); | Octagon (const Octagon & orig); |
| Площадь фигуры | double Square(); | double Square(); | double Square(); |
| Печать фигуры | void Print(); | void Print(); | void Print(); |
| Деконструктор класса | ~ Pentagon (); | ~ hexagon (); | ~Octagon(); |

**figure.h**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif // !FIGURE\_H

#endif

***Pentagon.h***

#ifndef PENTAGON\_H

#define PENTAGON\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Pentagon :

public Figure

{

public:

Pentagon();

Pentagon(size\_t side\_a);

Pentagon(std::istream &is);

Pentagon(const Pentagon& orig);

double Square() override;

void Print() override;

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

virtual ~Pentagon();

private:

size\_t side\_a;

size\_t side\_b;

};

#endif

***Octagon.h***

#ifndef OCTAGON\_H

#define OCTAGON\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Octagon :

public Figure

{

public:

Octagon();

Octagon(double side\_a, double side\_b);

Octagon(std::istream &is);

Octagon(const Octagon& orig);

double Square() override;

void Print() override;

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

virtual ~Octagon();

private:

double side\_a;

double side\_b;

};

#endif // !HEXAGON.H

***Hexagon.h***

#ifndef HEXAGON\_H

#define HEXAGON\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Hexagon :

public Figure

{

public:

Hexagon();

Hexagon(size\_t side\_a);

Hexagon(std::istream &is);

Hexagon(const Hexagon& orig);

double Square() override;

void Print() override;

virtual ~Hexagon();

private:

size\_t side\_a;

};

#endif // !HEXAGON\_H

***Lab\_w1.cpp***

#include <cstdlib>

#include "Pentagon.h"

#include "Octagon.h"

#include "Hexagon.h"

// Для правильного пятиугольника задаётся радиус описанной окружности.

// Для восьмиугольника задаётся меньшая из диагоналей(ширина восьмиугольника) и сторона.

// Для правильного шестиугольника нужно задать только одну сторону.

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

{

std::cout << "Pentagon(write radius)" << std::endl;

Figure \*pentagon = new Pentagon(std::cin);

pentagon->Print();

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

delete pentagon;

std::cout << "Octagon(write diag and length)" << std::endl;

Figure \*octagon = new Octagon(std::cin);

octagon->Print();

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

delete octagon;

std::cout << "Hexagon(write length)" << std::endl;

Figure \*hexagon = new Hexagon(std::cin);

hexagon->Print();

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

delete hexagon;

system("PAUSE");

return 0;

}

***Pentagon.cpp***

#include <iostream>

#include <cmath>

#include "Pentagon.h"

Pentagon::Pentagon() : Pentagon(0)

{

}

Pentagon::Pentagon(size\_t i) : side\_a(i)

{

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

}

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

{

is >> side\_a;

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

}

Pentagon::Pentagon(const Pentagon& orig)

{

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

side\_a = orig.side\_a;

}

double Pentagon::Square()

{

std::cout << "sq = ";

return (5 / 2) \* side\_a \* side\_a \* sin(72);

}

void Pentagon::Print()

{

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

}

Pentagon::~Pentagon()

{

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

}

***Octagon.cpp***

#include <iostream>

#include <cmath>

#include "Octagon.h"

Octagon::Octagon():Octagon(0, 0)

{

}

Octagon::Octagon(size\_t i, size\_t j) : side\_a(i), side\_b(j)

{

std::cout << "Octagon created: " << side\_a << ", " << side\_b << std::endl;

}

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

{

is >> side\_a;

is >> side\_b;

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

}

Octagon::Octagon(const Octagon& orig)

{

side\_a = orig.side\_a;

side\_b = orig.side\_b;

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

}

double Octagon::Square()

{

std::cout << "sq = ";

return 2 \* side\_a \* side\_b;

}

void Octagon::Print()

{

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

}

Octagon::~Octagon()

{

std::cout << "Octagon deleted." << std::endl;

}

***Hexagon.cpp***

#include <iostream>

#include <cmath>

#include "Hexagon.h"

Hexagon::Hexagon() : Hexagon(0)

{

}

Hexagon::Hexagon(size\_t i) : side\_a(i)

{

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

}

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

{

is >> side\_a;

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

}

Hexagon::Hexagon(const Hexagon& orig)

{

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

side\_a = orig.side\_a;

}

double Hexagon::Square()

{

std::cout << "sq = ";

return (3\* sqrt(3) \* side\_a \* side\_a) / 2;

}

void Hexagon::Print()

{

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

}

Hexagon::~Hexagon()

{

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

}

* **ВЫВОД КОНСОЛИ**

Pentagon(write radius)

2

Pentagon created:

a = 2

sq = 2.03059

Pentagon deleted.

Octagon(write diag and length)

3

2

Octagon created:

a = 3 b = 2

sq = 12

Octagon deleted.

Hexagon(write length)

2

Hexagon created:

a = 2

sq = 10.3923

Hexagon deleted.

Для продолжения нажмите любую клавишу . . .

* **ВЫВОДЫ**

Данная лабораторная работа является своеобразным вводным уроком в ООП. Так же происходит знакомство и с классами, перегрузками, деструкторами. В результате выполнения задания были спроектированы классы фигур, в которых использовались перегруженные операторы, дружественные функции и операции ввода-вывода из стандартных библиотек.

**Лабораторная работа №2**

* **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.

Контейнер первого уровня: список.

* **ОПИСАНИЕ**

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память следует распределять во время выполнения программы по мере необходимости отдельными блоками. Блоки связываются друг с другом с помощью указателей. Такой способ организации данных называется **динамической структурой данных**, поскольку она размещается в динамической памяти и ее размер изменяется во время выполнения программы.

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый динамическим распределением памяти, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

1. нет имени;

2. Для неё выделяется память в процессе выполнения программы;

3. Нефиксированное кол-во элементов структуры;

4. Размерность структуры может меняться в процессе выполнения программы;

5. В процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная типа **указатель** (ее значение – адрес этого объекта), посредством которой осуществляется доступ к динамической структуре.

Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.

Аргументы могут передаваться **по значению** и по ссылке. При передаче аргументов по значению внешний объект, который передается в качестве аргумента в функцию, не может быть изменен в этой функции. В функцию передается само значение этого объекта.

* **ИСХОДНЫЙ КОД**

|  |  |
| --- | --- |
| Pentagon.cpp | |
| Pentagon(); | Конструктор класса |
| Pentagon (std::istream &is); | Конструктор класса из стандартного потока |
| Pentagon (const Rectangle& orig); | Конструктор копии класса |
| double Square(); | Площадь фигуры |
| ~ Pentagon (); | Деконструктор класса |
| bool operator ==(const Pentagon &obj)  const; | Переопределенный оператор сравнения |
| Pentagon & operator =(const Pentagon  &obj); | Переопределенный оператор копирования |
| friend std::ostream& operator  (std::ostream &os, const Pentagon  &obj); | Переопределенный оператор вывода в поток std::ostream |
| friend std::istream& operator  (std::istream &is, Rectangle &obj); | Переопределенный оператор ввода из std::istream |
| void Print(); | Печать фигуры |

|  |  |
| --- | --- |
| TreeItem.cpp | |
| TBinaryTreeItem(); | Конструктор класса |
| int Side(); | Сторона фигуры |
| Hexagon GetHexagon(); | Возвращает фигуру из вершины |
| ~TBinaryTreeItem(); | Деструктор |

|  |  |
| --- | --- |
| Tree.cpp | |
| Tree(); | Конструктор класса |
| TreeItem\* find(int a); | Поиск вершины |
| void remove(int a); | Удаление вершины |
| void insert(Hexagon & & hexagon); | Вставка вершины |
| void print(); | Печать дерева |
| bool empty(); operator(std::ostream &os, virtual ~TBinaryTree(); | Проверка на пустое дерево поток  Деструктор |

***Tree.h***

#pragma once

#include "TreeItem.h"

class Tree

{

public:

Tree();

TreeItem\* insert(TreeItem \*node, size\_t key);

void remove(size\_t key);

void remove(TreeItem\* tree, size\_t key);

bool empty();

TreeItem\* find(TreeItem\* tree, size\_t key);

void print();

void print(TreeItem \*tree, size\_t depth);

Hexagon data(size\_t key);

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

~Tree();

private:

TreeItem \*root;

};

***TreeItem.h***

#pragma once

#include "Hexagon.h"

class TreeItem

{

public:

TreeItem(const Hexagon& data);

TreeItem(const TreeItem& orig);

TreeItem\* getSon();

TreeItem\* getSibling();

TreeItem\* setSon(TreeItem\* son);

TreeItem\* setSibling(TreeItem\* sibling);

size\_t getData();

Hexagon TreeItem::getHexagon() const;

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

~TreeItem();

private:

TreeItem \*sibling;

TreeItem \*son;

Hexagon data;

};

***Tree.cpp***

#include "stdafx.h"

#include "Tree.h"

Tree::Tree()

{

this->root = nullptr;

}

TreeItem \* Tree::insert(TreeItem \*node, size\_t key)

{

if (this->empty()) {

this->root = node;

std::cout << "Tree was empty. Item was setted as root." << std::endl;

return this->root;

}

TreeItem \* parent = find(this->root, key);

if (!parent) {

std::cout << "Parent with this square not found. Automatic insertion to the nearest place." << std::endl;

if (root->getSon()) {

TreeItem \*tmp = root->getSon();

while (tmp->getSibling())

tmp = tmp->getSibling();

tmp->setSibling(node);

return tmp->getSibling();

}

else {

root->setSon(node);

return root->getSon();

}

}

if (parent->getSon()) {

TreeItem \*tmp = parent->getSon();

while (tmp->getSibling())

tmp = tmp->getSibling();

tmp->setSibling(node);

return tmp->getSibling();

}

else {

parent->setSon(node);

return parent->getSon();

}

}

Hexagon Tree::data(size\_t key)

{

TreeItem \* tmp = this->find(this->root, key);

Hexagon \* shit = new Hexagon(0);

if (!tmp) {

return \*shit;

}

return tmp->getHexagon();

}

void Tree::remove(size\_t key)

{

if (root->getData() == key) {

delete root;

root = nullptr;

}

else {

remove(root, key);

}

}

void Tree::remove(TreeItem\* tree, size\_t key)

{

if (tree->getSon()) {

if (tree->getSon()->getData() == key) {

TreeItem \*tr = tree->getSon();

tree->setSon(tree->getSon()->getSibling());

tr->setSibling(nullptr);

delete tr;

return;

}

else {

remove(tree->getSon(), key);

}

}

if (tree->getSibling()) {

if (tree->getSibling()->getData() == key) {

TreeItem \*tr = tree->getSibling();

tree->setSibling(tree->getSibling()->getSibling());

tr->setSibling(nullptr);

delete tr;

return;

}

else {

remove(tree->getSibling(), key);

}

}

}

bool Tree::empty()

{

return this->root == nullptr;

}

TreeItem\* Tree::find(TreeItem\* tree, size\_t key)

{

TreeItem \*tr = nullptr;

if (tree->getData() == key)

return tree;

if (tree->getSon()) {

tr = find(tree->getSon(), key);

if (tr != nullptr) {

return tr;

}

}

if (tree->getSibling()) {

tr = find(tree->getSibling(), key);

if (tr != nullptr) {

return tr;

}

}

return nullptr;

}

void Tree::print()

{

this->print(this->root, 0);

}

void Tree::print(TreeItem \*tree, size\_t depth)

{

if (tree)

for (size\_t i = 0; i < depth; i++)

std::cout << "\t";

//tree->getHexagon().print();

std::cout << tree->getHexagon() << std::endl;

if (tree->getSon())

print(tree->getSon(), depth + 1);

if (tree->getSibling())

print(tree->getSibling(), depth);

}

Tree::~Tree()

{

delete root;

}

std::ostream & operator<<(std::ostream & os, Tree & obj)

{

obj.print();

return os;

}

***TreeItem.cpp***

#include "stdafx.h"

#include "TreeItem.h"

#include <iostream>

TreeItem::TreeItem(const Hexagon& hexagon)

{

this->data = hexagon;

this->sibling = nullptr;

this->son = nullptr;

}

TreeItem::TreeItem(const TreeItem& orig)

{

this->data = orig.data;

this->son = orig.son;

this->sibling = orig.sibling;

}

TreeItem \* TreeItem::getSon()

{

return this->son;

}

TreeItem \* TreeItem::getSibling()

{

return this->sibling;

}

TreeItem \* TreeItem::setSon(TreeItem\* son)

{

this->son = son;

return this;

}

TreeItem \* TreeItem::setSibling(TreeItem\* sibling)

{

this->sibling = sibling;

return this;

}

Hexagon TreeItem::getHexagon() const

{

return data;

}

size\_t TreeItem::getData()

{

return data.Square();

}

TreeItem::~TreeItem()

{

delete sibling;

delete son;

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

}

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

{

os << "[" << obj.getHexagon() << "]" << std::endl;

return os;

}

* **ВЫВОД КОНСОЛИ**

Type 'h' or 'help' to get help.

H

h

'q' or 'quit' - exit the program.

'r' or 'remove s' - remove the hexagon with area s.

'd' or 'destroy' - delete the tree.

'p' or 'print' - output the tree.

'ins' or 'insert s' - insert a hexagon into the tree.

'h 'or' help '- display the help.

ins

2 3

Tree was empty. Item was setted as root.

p

a = 2 square = 10.3923

d

The tree was deleted.

p

The tree is empty.

ins

4 4

Tree was empty. Item was setted as root.

p

a = 4 square = 41.5692

q

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

Данная лабораторная работа обеспечивает навыки работы с динамическими структурами, которые применимы, когда используются переменные, имеющие довольно большой размер , необходимые в одних частях программы и совершенно не нужные в других; в процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем; когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

**Лабораторная работа №3**

* **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.

* **ОПИСАНИЕ**

**Smart pointer** — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

В новом стандарте появились следующие умные указатели: unique\_ptr, shared\_ptr и weak\_ptr. Все они объявлены в заголовочном файле <memory>.

**1. unique\_ptr**

Этот указатель пришел на смену старому и проблематичному auto\_ptr. Основная проблема последнего заключается в правах владения. Объект этого класса теряет права владения ресурсом при копировании (присваивании, использовании в конструкторе копий, передаче в функцию по значению).Это очень неудобно, при работе с контейнером из умных указателей.В отличии от auto\_ptr, unique\_ptr запрещает копирование.

**2. shared\_ptr**

Это самый популярный и самый широкоиспользуемый умный указатель. Он начал своё развитие как часть библиотеки boost. Данный указатель был столь успешным, что его включили в C++ Technical Report 1 и он был доступен в пространстве имен tr1 — std::tr1::shared\_ptr<>. В отличии от рассмотренных выше указателей, shared\_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0. Как видно, система реализует одно из основных правил сборщика мусора.

**3. weak\_ptr**

Этот указатель также, как и shared\_ptr начал свое рождение в проекте boost, затем был включен в C++ Technical Report 1 и, наконец, пришел в новый стандарт.

Данный класс позволяет разрушить циклическую зависимость, которая, несомненно, может образоваться при использовании shared\_ptr.

* **ИСХОДНЫЙ КОД**

|  |  |
| --- | --- |
| TreeItem.cpp | |
| TreeItem(const std::sharedptr &obj); | Конструктор класса |
| std::sharedptr GetFigure(); | Получение фигуры |
| ~TBinaryTreeItem(); | Деструктор класса |

|  |  |
| --- | --- |
| Tree.cpp | |
| Tree(); | Конструктор класса |
| std::sharedptr find(std::sharedptr &obj); | Поиск вершины |
| void remove(int32 side); | Удаление вершины |
| void insert(std::sharedptr &obj); | Вставка вершины |
| void print(); | Печать дерева |
| bool empty(); | Проверка пустое ли дерево |
| virtual ~TBinaryTree(); | Деструктор класса |

**TreeItem.h**

#pragma once

#include "Hexagon.h"

#include "Octagon.h"

#include "Pentagon.h"

#include <memory>

class TreeItem

{

public:

TreeItem(const std::shared\_ptr<Hexagon>& hexagon, const std::shared\_ptr<Octagon>& octagon, const std::shared\_ptr<Pentagon>& pentagon, size\_t key);

TreeItem(const TreeItem& orig);

std::shared\_ptr<TreeItem> GetSon();

std::shared\_ptr<TreeItem> GetSibling();

void SetSon(std::shared\_ptr<TreeItem> son);

void SetSibling(std::shared\_ptr<TreeItem> sibling);

size\_t GetKey() const;

std::shared\_ptr<Hexagon> GetHexagon() const;

std::shared\_ptr<Pentagon> GetPentagon() const;

std::shared\_ptr<Octagon> GetOctagon() const;

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

~TreeItem();

private:

std::shared\_ptr<TreeItem> son;

std::shared\_ptr<TreeItem> sibling;

std::shared\_ptr<Hexagon> hexagon;

std::shared\_ptr<Pentagon> pentagon;

std::shared\_ptr<Octagon> octagon;

size\_t key;

};

**Tree.h**

#pragma once

#include "TreeItem.h"

class Tree

{

public:

Tree();

std::shared\_ptr<TreeItem> insert(std::shared\_ptr<TreeItem> node, size\_t key);

void remove(size\_t key);

void remove(std::shared\_ptr<TreeItem> tree, size\_t key);

bool empty();

std::shared\_ptr<TreeItem> find(std::shared\_ptr<TreeItem> tree, size\_t key);

void Print();

void Print(std::shared\_ptr<TreeItem> tree, size\_t depth);

std::shared\_ptr<Hexagon> hexagon(size\_t key);

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

~Tree();

private:

std::shared\_ptr<TreeItem> root;

};

**TreeItem.cpp**

#include "stdafx.h"

#include "TreeItem.h"

#include <iostream>

TreeItem::TreeItem(const std::shared\_ptr<Hexagon>& hexagon, const std::shared\_ptr<Octagon>& octagon, const std::shared\_ptr<Pentagon>& pentagon, size\_t key)

{

this->hexagon = hexagon;

this->octagon = octagon;

this->pentagon = pentagon;

this->sibling = nullptr;

this->son = nullptr;

this->key = key;

}

TreeItem::TreeItem(const TreeItem& orig)

{

this->hexagon = orig.hexagon;

this->son = orig.son;

this->sibling = orig.sibling;

}

std::shared\_ptr<TreeItem> TreeItem::GetSon()

{

return this->son;

}

std::shared\_ptr<TreeItem> TreeItem::GetSibling()

{

return this->sibling;

}

void TreeItem::SetSon(std::shared\_ptr<TreeItem> son)

{

this->son = son;

}

void TreeItem::SetSibling(std::shared\_ptr<TreeItem> sibling)

{

this->sibling = sibling;

}

std::shared\_ptr<Hexagon> TreeItem::GetHexagon() const

{

return this->hexagon;

}

std::shared\_ptr<Pentagon> TreeItem::GetPentagon() const

{

return this->pentagon;

}

std::shared\_ptr<Octagon> TreeItem::GetOctagon() const

{

return this->octagon;

}

size\_t TreeItem::GetKey() const

{

return key;

}

TreeItem::~TreeItem()

{

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

}

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

{

os << "[" << \*obj.GetHexagon() << " " <<\*obj.GetOctagon() << " " <<\*obj.GetPentagon() << "]" << ":"<< obj.GetKey() << std::endl;

return os;

}

**Tree.cpp**

#include "stdafx.h"

#include "Tree.h"

Tree::Tree()

{

this->root = nullptr;

}

std::shared\_ptr<TreeItem> Tree::insert(std::shared\_ptr<TreeItem> node, size\_t key)

{

if (this->empty()) {

this->root = node;

std::cout << "Tree was empty. Item was setted as root." << std::endl;

return this->root;

}

std::shared\_ptr<TreeItem> parent = find(this->root, key);

if (!parent) {

std::cout << "Parent with this square not found. Automatic insertion to the nearest place." << std::endl;

if (root->GetSon()) {

std::shared\_ptr<TreeItem> tmp = root->GetSon();

while (tmp->GetSibling())

tmp = tmp->GetSibling();

tmp->SetSibling(node);

return tmp->GetSibling();

}

else {

root->SetSon(node);

return root->GetSon();

}

}

if (parent->GetSon()) {

std::shared\_ptr<TreeItem> tmp = parent->GetSon();

while (tmp->GetSibling())

tmp = tmp->GetSibling();

tmp->SetSibling(node);

return tmp->GetSibling();

}

else {

parent->SetSon(node);

return parent->GetSon();

}

}

std::shared\_ptr<Hexagon> Tree::hexagon(size\_t key)

{

std::shared\_ptr<TreeItem> tmp = this->find(this->root, key);

return tmp->GetHexagon();

}

void Tree::remove(size\_t key)

{

if (root->GetKey() == key) {

root = nullptr;

}

else {

remove(root, key);

}

}

void Tree::remove(std::shared\_ptr<TreeItem> tree, size\_t key)

{

if (tree->GetSon()) {

if (tree->GetSon()->GetKey() == key) {

std::shared\_ptr<TreeItem> tr = tree->GetSon();

tree->SetSon(tree->GetSon()->GetSibling());

tr->SetSibling(nullptr);

return;

}

else {

remove(tree->GetSon(), key);

}

}

if (tree->GetSibling()) {

if (tree->GetSibling()->GetKey() == key) {

std::shared\_ptr<TreeItem> tr = tree->GetSibling();

tree->SetSibling(tree->GetSibling()->GetSibling());

tr->SetSibling(nullptr);

return;

}

else {

remove(tree->GetSibling(), key);

}

}

}

bool Tree::empty()

{

return this->root == nullptr;

}

std::shared\_ptr<TreeItem> Tree::find(std::shared\_ptr<TreeItem> tree, size\_t key)

{

std::shared\_ptr<TreeItem> tr = nullptr;

if (tree->GetKey() == key)

return tree;

if (tree->GetSon()) {

tr = find(tree->GetSon(), key);

if (tr != nullptr) {

return tr;

}

}

if (tree->GetSibling()) {

tr = find(tree->GetSibling(), key);

if (tr != nullptr) {

return tr;

}

}

return nullptr;

}

void Tree::Print()

{

this->Print(this->root, 0);

}

void Tree::Print(std::shared\_ptr<TreeItem> tree, size\_t depth)

{

if (tree)

for (size\_t i = 0; i < depth; i++)

std::cout << "\t";

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

if (tree->GetSon())

Print(tree->GetSon(), depth + 1);

if (tree->GetSibling())

Print(tree->GetSibling(), depth);

}

Tree::~Tree()

{

}

std::ostream & operator<<(std::ostream & os, Tree & obj)

{

obj.Print();

return os;

}

* **ВЫВОД КОНСОЛИ**

Type 'h' or 'help' to get help.

h

'q' or 'quit' - exit the program.

'r' or 'remove s' - remove the rectangle with area s.

'd' or 'destroy' - delete the tree.

'p' or 'print' - output the tree.

'ins' or 'insert s' - insert a rectangle into the tree.

'h 'or' help '- display the help.

ins

Enter 1 Hexagon side, 2 octagon sides, 1 pentagon sides, element key, parent key:

1 2 2 2 4 2

Tree was empty. Item was setted as root.

p

[Hexagon with a = 1, Square = 2.59808

Octagon with a = 2, b = 2, Square = 2 Pentagon with side a = 2, Square = 2.03059]:4

d

The tree was deleted.

ins

Enter 1 Hexagon side, 2 octagon sides, 1 pentagon sides, element key, parent key:

1 2 3 2 1 2

Tree was empty. Item was setted as root.

p

[Hexagon with a = 1, Square = 2.59808

Octagon with a = 2, b = 3, Square = 3 Pentagon with side a = 2, Square = 2.03059]:1

q

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

Умные указатели — очень удобная и полезная вещь. Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

**Лабораторная работа №4**

* **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.

* **ОПИСАНИЕ**

**Шаблон класса** начинается с ключевого слова template. В угловых скобках записывают параметры шаблона. При использовании шаблона на место этих параметров шаблону передаются аргументы: типы и константы, перечисленные через запятую.

Типы могут быть как стандартными, так и определенными пользователем. Для их описания в списке параметров используется ключевое слово class.

Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса.

Локальные классы не могут иметь шаблоны в качестве своих элементов.

Шаблоны методов не могут быть виртуальными.

Шаблоны классов могут содержать статические элементы, дружественные функции и классы.

Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.

* **ИСХОДНЫЙ КОД**

|  |  |
| --- | --- |
| * TreeItem.cpp | |
| TreeItem(const std::sharedptr &obj); Конструктор класса | Конструктор класса |
| std::sharedptr GetFigure(); | Получение фигуры |
| TreeItem(); | Деструктор класса |

|  |  |
| --- | --- |
| Tree.cpp | |
| std::sharedptr<TreeItem &obj); | Поиск вершины |
| void insert(std::sharedptr &obj; | Вставка вершины |

***Tree.h***

#pragma once

#include "TreeItem.h"

#include <memory>

template <class T> class Tree

{

public:

Tree();

std::shared\_ptr<TreeItem<T>> insert(std::shared\_ptr<TreeItem<T>> node, size\_t key);

void remove(size\_t key);

void remove(std::shared\_ptr<TreeItem<T>> tree, size\_t key);

bool empty();

std::shared\_ptr<TreeItem<T>> find(std::shared\_ptr<TreeItem<T>> tree, size\_t key);

void Print();

void Print(std::shared\_ptr<TreeItem<T>> tree, size\_t depth);

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

~Tree();

private:

std::shared\_ptr<TreeItem<T>> root;

};

***TreeItem.h***

#pragma once

#include "Hexagon.h"

#include "Pentagon.h"

#include "Octagon.h"

#include <memory>

template <class T> class TreeItem

{

public:

TreeItem(const std::shared\_ptr<T>& figure, size\_t key);

TreeItem(const TreeItem& orig);

std::shared\_ptr<TreeItem<T>> GetSon();

std::shared\_ptr<TreeItem<T>> GetSibling();

void SetSon(std::shared\_ptr<TreeItem<T>> son);

void SetSibling(std::shared\_ptr<TreeItem<T>> sibling);

size\_t GetKey() const;

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

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

~TreeItem();

private:

std::shared\_ptr<TreeItem<T>> son;

std::shared\_ptr<TreeItem<T>> sibling;

std::shared\_ptr<T> figure;

size\_t key;

};

***Tree.cpp***

#include "stdafx.h"

#include "Tree.h"

template <class T>

Tree<T>::Tree()

{

this->root = nullptr;

}

template <class T>

std::shared\_ptr<TreeItem<T>> Tree<T>::insert(std::shared\_ptr<TreeItem<T>> node, size\_t key)

{

if (this->empty()) {

this->root = node;

std::cout << "Tree was empty. Item was setted as root." << std::endl;

return this->root;

}

std::shared\_ptr<TreeItem<T>> parent = find(this->root, key);

if (!parent) {

std::cout << "Parent with this key not found. Automatic insertion to the nearest place." << std::endl;

if (root->GetSon()) {

std::shared\_ptr<TreeItem<T>> tmp = root->GetSon();

while (tmp->GetSibling())

tmp = tmp->GetSibling();

tmp->SetSibling(node);

return tmp->GetSibling();

}

else {

root->SetSon(node);

return root->GetSon();

}

}

if (parent->GetSon()) {

std::shared\_ptr<TreeItem<T>> tmp = parent->GetSon();

while (tmp->GetSibling())

tmp = tmp->GetSibling();

tmp->SetSibling(node);

return tmp->GetSibling();

}

else {

parent->SetSon(node);

return parent->GetSon();

}

}

template <class T>

void Tree<T>::remove(size\_t key)

{

if (root->GetKey() == key) {

root = nullptr;

}

else {

remove(root, key);

}

}

template <class T>

void Tree<T>::remove(std::shared\_ptr<TreeItem<T>> tree, size\_t key)

{

if (tree->GetSon()) {

if (tree->GetSon()->GetKey() == key) {

std::shared\_ptr<TreeItem<T>> tr = tree->GetSon();

tree->SetSon(tree->GetSon()->GetSibling());

tr->SetSibling(nullptr);

return;

}

else {

remove(tree->GetSon(), key);

}

}

if (tree->GetSibling()) {

if (tree->GetSibling()->GetKey() == key) {

std::shared\_ptr<TreeItem<T>> tr = tree->GetSibling();

tree->SetSibling(tree->GetSibling()->GetSibling());

tr->SetSibling(nullptr);

return;

}

else {

remove(tree->GetSibling(), key);

}

}

}

template <class T>

bool Tree<T>::empty()

{

return this->root == nullptr;

}

template <class T>

std::shared\_ptr<TreeItem<T>> Tree<T>::find(std::shared\_ptr<TreeItem<T>> tree, size\_t key)

{

std::shared\_ptr<TreeItem<T>> tr = nullptr;

if (tree->GetKey() == key)

return tree;

if (tree->GetSon()) {

tr = find(tree->GetSon(), key);

if (tr != nullptr) {

return tr;

}

}

if (tree->GetSibling()) {

tr = find(tree->GetSibling(), key);

if (tr != nullptr) {

return tr;

}

}

return nullptr;

}

template <class T>

void Tree<T>::Print()

{

this->Print(this->root, 0);

}

template <class T>

void Tree<T>::Print(std::shared\_ptr<TreeItem<T>> tree, size\_t depth)

{

if (tree)

for (size\_t i = 0; i < depth; i++)

std::cout << "\t";

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

if (tree->GetSon())

Print(tree->GetSon(), depth + 1);

if (tree->GetSibling())

Print(tree->GetSibling(), depth);

}

template <class T>

Tree<T>::~Tree()

{

}

template <class T>

std::ostream & operator<<(std::ostream & os, Tree<T> & obj)

{

obj.Print();

return os;

}

#include "Tree.h"

template class Tree<Hexagon>;

template std::ostream& operator<<(std::ostream& os, const Tree<Hexagon>& tree);

***TreeItem.cpp***

#include "stdafx.h"

#include "TreeItem.h"

#include <iostream>

template<class T>

TreeItem<T>::TreeItem(const std::shared\_ptr<T>& figure, size\_t key)

{

this->figure = figure;

this->key = key;

this->sibling = nullptr;

this->son = nullptr;

}

template <class T>

TreeItem<T>::TreeItem(const TreeItem<T>& orig)

{

this->figure = orig.figure;

this->son = orig.son;

this->sibling = orig.sibling;

}

template <class T>

std::shared\_ptr<TreeItem<T>> TreeItem<T>::GetSon()

{

return this->son;

}

template <class T>

std::shared\_ptr<TreeItem<T>> TreeItem<T>::GetSibling()

{

return this->sibling;

}

template <class T>

void TreeItem<T>::SetSon(std::shared\_ptr<TreeItem<T>> son)

{

this->son = son;

}

template <class T>

void TreeItem<T>::SetSibling(std::shared\_ptr<TreeItem<T>> sibling)

{

this->sibling = sibling;

}

template <class T>

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

{

return this->figure;

}

template <class T>

size\_t TreeItem<T>::GetKey() const

{

return key;

}

template <class T>

TreeItem<T>::~TreeItem()

{

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

}

template <class T>

std::ostream& operator<<(std::ostream& os, const TreeItem<T>& obj)

{

os << "[" << \*obj.GetFigure() << "]" << ":"<< obj.GetKey() << std::endl;

return os;

}

#include "Hexagon.h"

template class TreeItem<Hexagon>;

template std::ostream& operator<<(std::ostream& os, const TreeItem<Hexagon>& obj);

* **ВЫВОД КОНСОЛИ**

Type 'h' or 'help' to get help.

h

'q' or 'quit' - exit the program.

'r' or 'remove s' - remove the rectangle with area s.

'd' or 'destroy' - delete the tree.

'p' or 'print' - output the tree.

'ins' or 'insert s' - insert a rectangle into the tree.

'h 'or' help '- display the help.

ins

Enter 1 hexagon sid, element key, parent key:

1 2 3

Tree was empty. Item was setted as root.

p

[a = 1 Square = 2.59808]:2

r

2

p

The tree is empty.

ins

Enter 1 hexagon sid, element key, parent key:

1 2 3

Tree was empty. Item was setted as root.

p

[a = 1 Square = 2.59808]:2

r

2

p

The tree is empty.

ins

Enter 1 hexagon sid, element key, parent key:

4 2 1

Tree was empty. Item was setted as root.

p

[a = 4 Square = 41.5692]:2

q

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

Шаблон класса позволяет задать класс, параметризованный типом данных. Передача классу различных типов данных в качестве параметра создает семейство родственных классов. Наиболее широкое применение шаблоны находят при создании контейнерных классов. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

**Лабораторная работа №5**

* **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.

* **ОПИСАНИЕ**

Итератор — это объект, который может выполнять итерацию элементов в контейнере STL и предоставлять доступ к отдельным элементам. Все контейнеры STL предоставляют итераторы, чтобы алгоритмы могли получить доступ к своим элементам стандартным способом, независимо от типа контейнера, в котором сохранены элементы.  
Для получения итераторов контейнеры в C++ обладают такими функциями, как begin() и end(). Функция begin() возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов). Функция end() возвращает итератор, который указывает на следующую позицию после последнего элемента, то есть по сути на конец контейнера. Если контейнер пуст, то итераторы, возвращаемые обоими методами begin и end совпадают. Если итератор begin не равен итератору end, то между ними есть как минимум один элемент.  
Принцип работы итераторов очень похожий на работу указателей: для получения значения также используется оператор разыменования, операции инкремента и декремента обеспечивают доступ в прямом и обратном направлении соответственно.

* **ИСХОДНЫЙ КОД**

***TIterator.h***

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <memory>

#include <iostream>

template <class N, class T>

class TIterator

{

public:

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

cur = n;

}

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

return cur->GetFigure();

}

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

return cur->GetFigure();

}

void operator++() {

cur = cur->GetNext();

}

TIterator operator++ (int) {

TIterator cur(\*this);

++(\*this);

return cur;

}

bool operator== (const TIterator &i) {

return (cur == i.cur);

}

bool operator!= (const TIterator &i) {

return !(cur == i.cur);

}

private:

std::shared\_ptr<N> cur;

};

#endif

* **ВЫВОД КОНСОЛИ**

To access the menu, press 7

7

List of operation:

1) Add Pentagon

2) Add Octagon

3) Add Hexagon

4) Remove the pieces of wood on the maximum side

5) Print tree

6) Print tree with iterator

0) exit

1

2 2 2

6

Side\_A = 2, Side\_B = 2, Side\_C = 2

3

2

6

Side = 2

4

2

6

Side = 2

6

Side = 2

5

Side = 2

0

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или [списком](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Проектирование класса итератора обычно тесно связано с соответствующим классом контейнера. Обычно контейнер предоставляет методы создания итераторов. Итератор похож на [указатель](https://ru.wikipedia.org/wiki/%D0%A3%D0%BA%D0%B0%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) своими основными операциями: он указывает на отдельный элемент коллекции объектов (предоставляет доступ к элементу) и содержит функции для перехода к другому элементу списка (следующему или предыдущему). Контейнер, который реализует поддержку итераторов, должен предоставлять первый элемент списка, а также возможность проверить, перебраны ли все элементы контейнера (является ли итератор конечным). В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

**Лабораторная работа №6**

* **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.

Контейнер второго уровня: очередь.

* **ОПИСАНИЕ**

"Allocator" - переводится как "распределитель" - попросту говоря аллокатор - это действующая по определённой логике высокоуровневая прослойка между запросами памяти под динамические объекты и стандартными сервисами выделения памяти (new/malloc или другими (например запросами напрямую к ядру о.с.)), конечно же прослойка берет на себя и вопросы управлением отдачей уже ненужной памяти назад. По другому можно сказать - что аллокатор это реализация стратегии управления памятью.

Естественно что епархия аллокаторов - в основном объекты-контейнеры, т.к. именно там остро проявляется необходимость в некоторой стратегии, но так же можно использовать и для одиночных динамических объектов, на протяжении всего времени жизни программы размещая эти объекты в заранее выделенном бассейне "pool" памяти. Кстати обычно pool переводят как пул, но бассейн вроде звучит - так что бассейне памяти. Так же обычно pool аллокаторы - понятие, применяемое к определенной категории аллокаторов, работающих с объектами одного размера. Использование стратегии будет не выгодно по памяти, но очень выгодно по времени выполнения.

* **ИСХОДНЫЙ КОД**

|  |  |
| --- | --- |
| TAllocationBlock.cp | |
| TAllocationBlock(sizet size,sizet count); | Конструктор класса |
| void \*allocate(); | Выделение памяти |
| void deallocate(void \*pointer); | Освобождение памяти |
| bool Empty() | Проверка на пустоту |
| bool hasfreeblocks(); | Есть ли пустые блоки |
| virtual ~TAllocationBlock(); | Деструктор |

**TAllocationBlock.h**

#ifndef TALLOCATIONBLOCK\_H

#define TALLOCATIONBLOCK\_H

#include <cstdlib>

class TAllocationBlock {

public:

// Конструктор блока

TAllocationBlock(size\_t size, size\_t count);

// Аллоцирование под запрос

void \*allocate();

// Возврат памяти

void deallocate(void \*pointer);

// Есть ли свободные блоки?

bool has\_free\_blocks();

// Деконструктор блока

virtual ~TAllocationBlock();

private:

// сколько выделено

size\_t \_size;

// текущий размер

size\_t \_count;

// используемые блоки

char \*\_used\_blocks;

// свободные блоки

void \*\*\_free\_blocks;

// число свободных

size\_t \_free\_count;

};

#endif /\* TALLOCATIONBLOCK\_H \*/

**TAllocationBlock.cpp**

#include "TAllocationBlock.h"

#include <iostream>

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count) :

\_size(size), \_count(count) {

\_used\_blocks = (char\*)malloc(\_size\*\_count);

\_free\_blocks = (void\*\*)malloc(sizeof(void\*) \* \_count);

for (size\_t i = 0; i<\_count; i++) \_free\_blocks[i] = \_used\_blocks + i\*\_size;

\_free\_count = \_count;

std::cout << "TAllocationBlock: Memory init" << std::endl;

}

void \*TAllocationBlock::allocate() {

void \*result = nullptr;

if (\_free\_count > 0) {

result = \_free\_blocks[\_free\_count - 1];

\_free\_count--;

std::cout << "TAllocationBlock: Allocate " << (\_count - \_free\_count) <<

" of " << \_count << std::endl;

}

else {

std::cout << "TAllocationBlock: No memory exception kek" << std::endl;

}

return result;

}

void TAllocationBlock::deallocate(void \*pointer) {

std::cout << "TAllocationBlock: Deallocate block " << std::endl;

\_free\_blocks[\_free\_count] = pointer;

\_free\_count++;

}

bool TAllocationBlock::has\_free\_blocks() {

return \_free\_count>0;

}

TAllocationBlock::~TAllocationBlock() {

if (\_free\_count<\_count) {

std::cout << "TAllocationBlock: Memory leak?" << std::endl;

}

else {

std::cout << "TAllocationBlock: Memory freed" << std::endl;

}

free(\_free\_blocks);

free(\_used\_blocks);

}

* **ВЫВОД КОНСОЛИ**

Список операций:

1) Добавление треугольника

2) Добавление восьмиугольник

3) Добавление шестиугольника

4) Удаление фигуры из дерева по максимальной стороне

5) Печать дерева

6) Печать дерева c итератором

0) Выход

1

1

2

3

2

2

3

5

5

Side\_A = 1,Side\_B = 2,Side\_C = 3

Side = 2 Side = 5

TAllocationBlock: Memory freed ==2753== ==2753== HEAP SUMMARY: ==2753== in use at exit: 0 bytes in 0 blocks ==2753== total heap usage: 11 allocs,11 frees,82,304 bytes allocated ==2753== ==2753== All heap blocks were freed --no leaks are possible ==2753== ==2753== For counts of detected and suppressed errors,rerun with: -v ==2753== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

* **ВЫВОД**

Использование аллокаторов позволяет добиться существенного повышения производительности в работе с динамическими объектами (особенно с объектами-контейнерами). Если в коде много работы по созданию/уничтожению динамических объектов, и нет никакой стратегии управления памятью, а только использование стандартных сервисов new/malloc - то весьма вероятно, что не смотря на то, что программа написанна на Си++ (или даже чистом Си) - она окажется более медленной чем схожая программа написанная на Java или C#.

**Лабораторная работа №7**

* **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).

* **ОПИСАНИЕ**

**Принцип открытости-закрытости** – самый простой и очевидный принцип, гласящий: любые программные единицы (классы, структуры, модули) должны быть открыты для расширения и закрыты для изменения. Если класс уже был написан, одобрен, протестирован, возможно, внесён в библиотеку и включен в проект, после этого пытаться модифицировать его содержимое нельзя. Но никто не может запретить расширять его возможности другими доступными средствами. По сути, этот принцип просто предполагает грамотное использование двух принципов ООП: [абстракции](https://pro-prof.com/forums/topic/oop-abstraction) и [полиморфизма](https://pro-prof.com/forums/topic/oop-polymorphism).

**Контейнер** — это набор однотипных элементов. Здесь не случайно используется слово «набор», а не «множество», так как множество — это тоже контейнер. По этому определению массив — это контейнер. Каталог файлов на диске — тоже контейнер.

Каждый контейнер характеризуется, в первую очередь, своим именем и типом входящих в него элементов. Имя контейнера — это имя переменной в программе, которое подчиняется правилам видимости С++. Как объект, контейнер должен обладать временем жизни в зависимости от места и времени создания, причем время жизни контейнера в общем случае не зависит от времени жизни его элементов.

* **ИСХОДНЫЙ КОД**

|  |  |
| --- | --- |
| TTree.h | |
| void recRemByType(std::sharedptr&, int&); | Удаление всех фигур одного типа |
| void recInsert(std::sharedptr&, O&); | Вставка фигуры |
| void Print(); | Печать очереди |
| void recInorder(std::sharedptr&); | Печать |
| void recRemLesser(std::sharedptr&, int32t&); | Удаление по стороне одной фигуры |
| virtual ~TQueue(); | Деконструктор класса |

**TTree.h**

#ifndef TTREE\_H

#define TTREE\_H

#include "figure.h"

#include <memory>

#include <iostream>

template <class Q, class O> class TTree {

private:

class Node {

public:

Q data;

std::shared\_ptr<Node> son;

std::shared\_ptr<Node> sibling;

Node();

Node(O&);

int itemsInNode;

};

std::shared\_ptr<Node> root;

void recRemByType(std::shared\_ptr<Node>&, int&);

void recInsert(std::shared\_ptr<Node>&, O&);

void recInorder(std::shared\_ptr<Node>&);

void recRemLesser(std::shared\_ptr<Node>&, int32\_t&);

void clear(std::shared\_ptr<Node>&, std::shared\_ptr<Node>&);

void GG(std::shared\_ptr<Node>& node);

public:

TTree();

void insert(O&);

void inorder();

void removeByType(int&);

void removeLesser(int32\_t&);

void RM();

};

#endif

**TBTreeItem.h**

#ifndef TBINARYTREEITEM

#define TBINARYTREEITEM

#include <cstdlib>

#include <iostream>

#include <memory>

#include "hexagon.h"

#include "Pentagon.h"

#include "octagon.h"

template <class T> class TBinaryTree;

template <class T> class TBinaryTreeItem

{

public:

TBinaryTreeItem();

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

TBinaryTreeItem(const std::shared\_ptr<T> obj, std::shared\_ptr<TBinaryTreeItem<T>> l, std::shared\_ptr<TBinaryTreeItem<T>> r, std::shared\_ptr<TBinaryTreeItem<T>> p);

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

~TBinaryTreeItem();

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

friend class TBinaryTree<T>;

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

private:

std::shared\_ptr<T> item;

std::shared\_ptr<TBinaryTreeItem<T>> left;

std::shared\_ptr<TBinaryTreeItem<T>> right;

std::shared\_ptr<TBinaryTreeItem<T>> parent;

};

#endif

**TTree.cpp**

#include "TTree.h"

#include "TBTree.h"

template <class Q, class O>

TTree<Q, O>::TTree() {

root = std::make\_shared<Node>(Node());

root->son = std::make\_shared<Node>(Node());

}

template <class Q, class O>

TTree<Q, O>::Node::Node()

{

son = sibling = nullptr;

itemsInNode = 0;

}

template <class Q, class O>

TTree<Q, O>::Node::Node(O& item)

{

data.insert(item);

itemsInNode = 1;

}

template <class Q, class O>

void TTree<Q, O>::recRemByType(std::shared\_ptr<Node>& node, int& type)

{

if (node->itemsInNode) {

for (auto i : node->data) {

if (i->type() == type) {

node->data.remove(i->Side());

node->itemsInNode--;

}

}

if (node->sibling) {

recRemByType(node->sibling, type);

}

if (node->son) {

recRemByType(node->son, type);

}

}

}

template <class Q, class O>

void TTree<Q, O>::removeByType(int& type)

{

recRemByType(root->son, type);

}

template <class Q, class O>

void TTree<Q, O>::recInsert(std::shared\_ptr<Node>& node, O& item)

{

if (node->itemsInNode < 5) {

node->data.insert(item);

node->itemsInNode++;

}

else {

auto sibl = node;

for (int i = 0; i < 3; i++) {

if (!sibl->sibling) {

sibl->sibling = std::make\_shared<Node>(Node(item));

return;

}

if (sibl->sibling->itemsInNode < 5) {

recInsert(sibl->sibling, item);

return;

}

sibl = sibl->sibling;

}

if (node->son) {

recInsert(node->son, item);

}

else {

node->son = std::make\_shared<Node>(Node(item));

}

}

}

template <class Q, class O>

void TTree<Q, O>::insert(O& item)

{

recInsert(root->son, item);

}

template <class Q, class O>

void TTree<Q, O>::recInorder(std::shared\_ptr<Node>& node)

{

if (node->itemsInNode) {

for (auto i : node->data) {

i->Print();

}

std::cout << "\n";

if (node->sibling) {

recInorder(node->sibling);

}

if (node->son) {

recInorder(node->son);

}

}

}

template <class Q, class O>

void TTree<Q, O>::inorder()

{

if (root->son->son || root->son->sibling) {

clear(root->son, root);

}

recInorder(root->son);

}

template <class Q, class O>

void TTree<Q, O>::recRemLesser(std::shared\_ptr<Node>& node, int32\_t& side)

{

if (node->itemsInNode) {

for (auto i : node->data) {

if (i->Side() == side) {

node->data.remove(i->Side());

node->itemsInNode--;

}

}

if (node->sibling) {

recRemLesser(node->sibling, side);

}

if (node->son) {

recRemLesser(node->son, side);

}

}

}

template <class Q, class O>

void TTree<Q, O>::removeLesser(int32\_t& side)

{

recRemLesser(root->son, side);

}

template <class Q, class O>

void TTree<Q, O>::clear(std::shared\_ptr<Node>& node, std::shared\_ptr<Node>& parent)

{

if (node) {

if (!node->itemsInNode) {

auto orphan = node;

auto orphanPar = parent;

if (node->sibling) {

orphan = node->sibling;

orphanPar = node;

}

else if (node->son) {

orphan = node->sibling;

orphanPar = node;

}

while (orphan->sibling || orphan->son) {

orphanPar = orphan;

if (orphan->sibling) {

orphan = orphan->sibling;

}

else if (orphan->son) {

orphan = orphan->son;

}

}

if (orphanPar->sibling == orphan) {

std::swap(node->data, orphan->data);

node->itemsInNode = orphan->itemsInNode;

orphanPar->sibling = nullptr;

}

else if (orphanPar->son == orphan) {

std::swap(node->data, orphan->data);

node->itemsInNode = orphan->itemsInNode;

orphanPar->son = nullptr;

}

}

}

if (node) {

if (node->son) {

clear(node->son, node);

}

if (node->sibling) {

clear(node->sibling, node);

}

}

}

template <class Q, class O>

void TTree<Q, O>::GG(std::shared\_ptr<Node>& node)

{

if (node->itemsInNode) {

node->data.gg();

if (node->sibling) {

GG(node->sibling);

}

if (node->son) {

GG(node->son);

}

}

}

template <class Q, class O>

void TTree<Q, O>::RM()

{

GG(root->son);

}

#include "TBTree.h"

template class TTree<TBinaryTree<Figure>, std::shared\_ptr<Figure>>;

**TBTreeItem.cpp**

#include "TBTreeItem.h"

template <class T>

TBinaryTreeItem<T>::TBinaryTreeItem()

{

item = nullptr;

left = nullptr;

right = nullptr;

parent = nullptr;

}

template <class T>

TBinaryTreeItem<T>::TBinaryTreeItem(const std::shared\_ptr<T> obj, std::shared\_ptr<TBinaryTreeItem<T>> l, std::shared\_ptr<TBinaryTreeItem<T>> r, std::shared\_ptr<TBinaryTreeItem<T>> p)

{

item = obj;

left = l;

right = r;

parent = p;

}

template <class T>

TBinaryTreeItem<T>::~TBinaryTreeItem()

{

}

template <class T>

std::shared\_ptr<T> TBinaryTreeItem<T>::GetFigure()

{

return this->item;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetNext()

{

if (this->right) {

std::shared\_ptr<TBinaryTreeItem<T>> temp = this->right;

while (temp->left) {

temp = temp->left;

}

return temp;

}

std::shared\_ptr<TBinaryTreeItem<T>> succ = nullptr;

std::shared\_ptr<TBinaryTreeItem<T>> n = std::make\_shared<TBinaryTreeItem<T>>(\*this);

std::shared\_ptr<TBinaryTreeItem<T>> root = std::make\_shared<TBinaryTreeItem<T>>(\*this);

while (root->parent) {

root = root->parent;

}

while (root) {

if (n->item->Side() < root->item->Side()) {

succ = root;

root = root->left;

}

else if (n->item->Side() > root->item->Side()) {

root = root->right;

}

else {

break;

}

}

return succ;

}

#include "figure.h"

template class TBinaryTreeItem<Figure>;

//template std::ostream& operator<<(std::ostream &out, TBinaryTreeItem<Figure> &obj);

* **ВЫВОД КОНСОЛИ**

To access the menu, press 7

7

List of actions:

1) Add pentagon(3 sides)

2) Add octagon(1 side)

3) Add hexagon(1 side)

4) Delete all figure the same type of wood. 1)Triangle 2)Octagon 3)Hexagon

5) Remove the pieces of wood on the maximum side

6) Print tree

0) exit

1 2 2 2

2

3

3

1

6

Side = 1

Side\_A = 2, Side\_B = 2, Side\_C = 2

Side = 3

4

2

6

Side = 1

Side\_A = 2, Side\_B = 2, Side\_C = 2

0

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

В данной лабораторной работе требовалось создать контейнер на основе двух, вложенных друг в друга, это помогло ознакомиться с принипом ООП – ОСР.

**Лабораторная работа №8**

* **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.

* **ОПИСАНИЕ**

Параллельная программа содержит несколько процессов, работающих совместно над выполнением некоторой задачи. Каждый процесс — это последовательная, а точнее — последовательность операторов, выполняемых один за другим. Последовательнаяпрограмма имеет один поток управления, а параллельная — несколько.

Совместная работа процессов параллельной программы осуществляется с помощью их взаимодействия. Взаимодействие программируется с применением разделяемых переменных или пересылки сообщений. Если используются разделяемые переменные, то один процесс осуществляет запись в переменную, считываемую другим процессом. При пересылке сообщений один процесс отправляет сообщение, которое получает другой.

* **ИСХОДНЫЙ КОД**

**TBTree.cpp**

#include "TBTree.h"

template <class T>

TBinaryTree<T>::TBinaryTree()

{

head = nullptr;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTree<T>::find(std::shared\_ptr<T> &obj)

{

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

while (item != nullptr) {

if (item->item->Side() == obj->Side()) {

return item;

}

else if (item->item->Side() > obj->Side()) {

item = item->left;

}

else if (item->item->Side() < obj->Side()) {

item = item->right;

}

}

return nullptr;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTree<T>::minValueNode(std::shared\_ptr<TBinaryTreeItem<T>> root)

{

std::shared\_ptr<TBinaryTreeItem<T>> min = root;

while (min->left != nullptr) {

min = min->left;

}

return min;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTree<T>::Successor(std::shared\_ptr<TBinaryTreeItem<T>> node)

{

if (!node->right) {

return minValueNode(node->right);

}

std::shared\_ptr<TBinaryTreeItem<T>> parent = node->parent;

while (parent && node == parent->right) {

node = parent;

parent = parent->parent;

}

return parent;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTree<T>::deleteNode(std::shared\_ptr<TBinaryTreeItem<T>> root, int32\_t side)

{

if (root == nullptr) {

return root;

}

if (side < root->item->Side()) {

root->left = deleteNode(root->left, side);

}

else if (side > root->item->Side()) {

root->right = deleteNode(root->right, side);

}

else {

if (root->left == nullptr) {

std::shared\_ptr<TBinaryTreeItem<T>> temp = root->right;

std::shared\_ptr<TBinaryTreeItem<T>> parent = root->parent;

temp->parent = parent;

root->left = nullptr;

root->right = nullptr;

root->parent = nullptr;

return temp;

}

else if (root->right == nullptr) {

std::shared\_ptr<TBinaryTreeItem<T>> temp = root->left;

std::shared\_ptr<TBinaryTreeItem<T>> parent = root->parent;

temp->parent = parent;

root->left = nullptr;

root->right = nullptr;

root->parent = nullptr;

return temp;

}

std::shared\_ptr<TBinaryTreeItem<T>> temp = minValueNode(root->right);

root->item = temp->item;

root->right = deleteNode(root->right, temp->item->Side());

}

return root;

}

template <class T>

void TBinaryTree<T>::remove(int32\_t side)

{

head = TBinaryTree<T>::deleteNode(head, side);

}

template <class T>

void TBinaryTree<T>::insert(std::shared\_ptr<T> &obj)

{

if (head == nullptr) {

std::shared\_ptr<TBinaryTreeItem<T>> tmp = nullptr;

head = std::make\_shared<TBinaryTreeItem<T>>(obj, nullptr, nullptr, tmp);

return;

}

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

std::shared\_ptr<TBinaryTreeItem<T>> parent = head;

while (true) {

if (obj->Side() <= item->item->Side()) {

if (item->left == nullptr) {

item->left = std::make\_shared<TBinaryTreeItem<T>>(obj, nullptr, nullptr, parent);

break;

}

else {

parent = item->left;

item = item->left;

}

}

else {

if (item->right == nullptr) {

item->right = std::make\_shared<TBinaryTreeItem<T>>(obj, nullptr, nullptr, parent);

break;

}

else {

parent = item->right;

item = item->right;

}

}

}

}

template <class T>

void TBinaryTree<T>::print\_tree(std::shared\_ptr<TBinaryTreeItem<T>> item, int32\_t a, std::ostream& os)

{

for (int32\_t i = 0; i < a; i++) {

os << " ";

}

item->GetFigure()->Print();

if (item->left != nullptr) {

TBinaryTree<T>::print\_tree(item->left, a + 1, os);

}

else if (item->right != nullptr) {

for (int32\_t i = 0; i <= a; i++) {

os << " ";

}

os << "null" << std::endl;

}

if (item->right != nullptr) {

TBinaryTree<T>::print\_tree(item->right, a + 1, os);

}

else if (item->left != nullptr) {

for (int32\_t i = 0; i <= a; i++) {

os << " ";

}

os << "null" << std::endl;

}

}

template <class T>

void TBinaryTree<T>::print()

{

if (head != nullptr) {

TBinaryTree<T>::print\_tree(head, 0, std::cout);

}

}

template <class T>

void TBinaryTree<T>::print(std::ostream& os)

{

if (head != nullptr) {

TBinaryTree<T>::print\_tree(head, 0, os);

}

}

template <class T>

bool TBinaryTree<T>::empty()

{

return head == nullptr;

}

template <class T>

TBinaryTree<T>::~TBinaryTree()

{

}

template <class T>

std::ostream& operator<<(std::ostream& os, TBinaryTree<T> &tree)

{

tree.print(os);

return os;

}

template <class T>

TIterator<TBinaryTreeItem<T>, T> TBinaryTree<T>::begin()

{

if (!head || !head->left) {

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

}

std::shared\_ptr<TBinaryTreeItem<T>> tmp = head;

while (tmp->left) {

tmp = tmp->left;

}

return TIterator<TBinaryTreeItem<T>, T>(tmp);

}

template <class T>

TIterator<TBinaryTreeItem<T>, T> TBinaryTree<T>::end()

{

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

}

template <class T>

void TBinaryTree<T>::Balanced(std::shared\_ptr<T> \*&arr, int l, int r)

{

if (l > r) {

return;

}

int m = (l + r) / 2;

insert(arr[m]);

Balanced(arr, l, m - 1);

Balanced(arr, m + 1, r);

}

template <class T>

void TBinaryTree<T>::MySort(std::shared\_ptr<T> \*&arr, int l, int r)

{

int x = l + (r - l) / 2;

int i = l;

int j = r;

while (i <= j) {

while (arr[i]->Side() < arr[x]->Side()) {

i++;

}

while (arr[j]->Side() > arr[x]->Side()) {

j--;

}

if (i <= j) {

std::shared\_ptr<T> tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;

j--;

}

}

if (i < r) {

MySort(arr, i, r);

}

if (l < j) {

MySort(arr, l, j);

}

}

template <class T>

void TBinaryTree<T>::recClear(std::shared\_ptr<TBinaryTreeItem<T>> node)

{

if (!node) {

return;

}

if (node->left) {

recClear(node->left);

}

if (node->right) {

recClear(node->right);

}

node->parent = nullptr;

node->item = nullptr;

}

template <class T>

void TBinaryTree<T>::Clear()

{

if (!head) {

return;

}

if (head->left) {

recClear(head->left);

}

if (head->right) {

recClear(head->right);

}

}

template <class T>

void TBinaryTree<T>::rm(std::shared\_ptr<TBinaryTreeItem<T>> node)

{

if (!node) {

return;

}

node->parent = nullptr;

if (node->left) {

rm(node->left);

}

if (node->right) {

rm(node->right);

}

}

template <class T>

void TBinaryTree<T>::gg()

{

if (!head) {

return;

}

if (head->left) {

rm(head->left);

}

if (head->right) {

rm(head->right);

}

}

#include "figure.h"

template class TBinaryTree<Figure>;

template std::ostream& operator<<(std::ostream& os, TBinaryTree<Figure> &tree);

**TBTree.h**

#ifndef TBTREE\_H

#define TBTREE\_H

#include <cstdlib>

#include <iostream>

#include <memory>

#include <future>

#include <mutex>

#include <thread>

#include "TBTreeItem.h"

#include "TIterator.h"

template <class T> class TBinaryTree

{

public:

TBinaryTree();

std::shared\_ptr<TBinaryTreeItem<T>> find(std::shared\_ptr<T> &obj);

void remove(int32\_t side);

void insert(std::shared\_ptr<T> &obj);

void print();

void print(std::ostream& os);

template <class A> friend std::ostream& operator<<(std::ostream& os, TBinaryTree<A> &tree);

bool empty();

void Balanced(std::shared\_ptr<T> \*&arr, int l, int r);

void MySort(std::shared\_ptr<T> \*&arr, int l, int r);

void Sort(std::shared\_ptr<T> \*&arr, int l, int r);

void gg();

void Clear();

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

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

virtual ~TBinaryTree();

private:

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

void recClear(std::shared\_ptr<TBinaryTreeItem<T>> node);

void rm(std::shared\_ptr<TBinaryTreeItem<T>> node);

std::shared\_ptr<TBinaryTreeItem<T>> Successor(std::shared\_ptr<TBinaryTreeItem<T>> node);

std::shared\_ptr<TBinaryTreeItem<T>> minValueNode(std::shared\_ptr<TBinaryTreeItem<T>> root);

std::shared\_ptr<TBinaryTreeItem<T>> deleteNode(std::shared\_ptr<TBinaryTreeItem<T>> root, int32\_t side);

void print\_tree(std::shared\_ptr<TBinaryTreeItem<T>> item, int32\_t a, std::ostream& os);

};

#endif

* **ВЫВОД КОНСОЛИ**

To access the menu, press 9

9

List of actions:

1) Add Pentagon

2) Add octagon

3) Add Hexagon

4) Remove the pieces of wood on the maximum side

5) Print Tree

6) Print c tree iterator

7) Sorting tree

8) Balancing tree using a parallel sorting

0) exit

1

2 2 2

5

Side\_A = 2, Side\_B = 2, Side\_C = 2

7

Enter the number of vertices

3

Left to enter 3 vertices

2

Left to enter 2 vertices

2

Left to enter 1 vertices

2

5

Side\_A = 2, Side\_B = 2, Side\_C = 2

Side = 2

Side = 2

Side = 2

null

null

null

0

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

Параллельность повышает производительность системы из-за более эффективного расходования системных ресурсов. Например, во время ожидания появления данных по сети, вычислительная система может использоваться для решения локальных задач. Параллельность повышает отзывчивость приложения. Если один поток занят расчетом или выполнением каких-то запросов, то другой поток может реагировать на действия пользователя.

**Лабораторная работа №9**

* **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:

o Генерация фигур со случайным значением параметров;

o Печать контейнера на экран;

o 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.

* **ОПИСАНИЕ**

Лямбда — это более короткая форма записи функтора. Что-то вроде анонимного функтора. Написание функтора — простая задача, но, как ни крути, мы пишем лишнее и понижаем читаемость кода. Писать целый класс функтора только для того, чтобы применить единожды — это не самый лучший дизайн. Именно здесь и приходят на помощь лямбда-функции.

* **ИСХОДНЫЙ КОД**

**Main.cpp**

#include <iostream>

#include <cstdlib>

#include <cstring>

#include <memory>

#include <random>

#include <functional>

#include "TBTree.h"

#include "figure.h"

#include "hexagon.h"

#include "octagon.h"

#include "Pentagon.h"

#include "TTree.h"

using namespace std;

int main(void) {

TBinaryTree<Figure> ttree;

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

TTree<command> tree(4);

command cmdInsert = [&]() {

std::cout << "Command: Insert" << std::endl;

std::default\_random\_engine generator;

std::uniform\_int\_distribution<int> distribution(1, 10);

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

int side = distribution(generator);

if ((side % 2) == 0) {

std::shared\_ptr<Figure> ptr = std::make\_shared<Pentagon>(Pentagon(side, side, side));

if (ttree.find(ptr) == nullptr) {

ttree.insert(ptr);

}

}

else if ((side % 3) == 0) {

std::shared\_ptr<Figure> ptr = std::make\_shared<Octagon>(Octagon(side));

if (ttree.find(ptr) == nullptr) {

ttree.insert(ptr);

}

}

else {

std::shared\_ptr<Figure> ptr = std::make\_shared<Hexagon>(Hexagon(side));

if (ttree.find(ptr) == nullptr) {

ttree.insert(ptr);

}

}

}

};

command cmdPrint = [&]() {

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

for (auto i : ttree) {

i->Print();

}

};

command cmdRemove = [&]() {

std::cout << "Command: Remove" << std::endl;

std::default\_random\_engine generator;

std::uniform\_int\_distribution<int> distribution(1, 10);

int side = distribution(generator);

std::cout << "Lesser than " << side << std::endl;

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

for (auto iter : ttree) {

if (iter->Side() < side) {

ttree.remove(iter->Side());

break;

}

}

}

};

tree.insert(std::shared\_ptr<command>(&cmdInsert, [](command\*) {}));

tree.insert(std::shared\_ptr<command>(&cmdPrint, [](command\*) {}));

tree.insert(std::shared\_ptr<command>(&cmdRemove, [](command\*) {}));

tree.insert(std::shared\_ptr<command>(&cmdPrint, [](command\*) {}));

tree.inorder();

system("PAUSE");

return 0;

}

* **ВЫВОД КОНСОЛИ**

Command: Insert

Octagon created: 3

Octagon copy created

Octagon created: 3

Octagon copy created

Hexagon created: 5

Pentagon created: 6, 6, 6

Pentagon copy created

Hexagon created: 5

Pentagon created: 2, 2, 2

Pentagon copy created

Pentagon created: 10, 10, 10

Pentagon copy created

Pentagon created: 6, 6, 6

Pentagon copy created

Octagon created: 9

Octagon copy created

Pentagon created: 4, 4, 4

Pentagon copy created

Command: Print

Side\_A = 2, Side\_B = 2, Side\_C = 2

Side = 3

Side\_A = 4, Side\_B = 4, Side\_C = 4

Side = 5

Side\_A = 6, Side\_B = 6, Side\_C = 6

Side = 9

Side\_A = 10, Side\_B = 10, Side\_C = 10

Command: Remove

Lesser than 3

Command: Print

Side = 3

Side\_A = 4, Side\_B = 4, Side\_C = 4

Side = 5

Side\_A = 6, Side\_B = 6, Side\_C = 6

Side = 9

Side\_A = 10, Side\_B = 10, Side\_C = 10

Для продолжения нажмите любую клавишу . . .

* **ВЫВОД**

В ходе выполнения последней лабораторной работы были получены основные навыки работы с лямбда выражениями, они, как и функциональные объекты позволяют хранить состояния, но их компактный синтаксит в отличие от функциональных объектов не требует объявления класса, а также лямбда-выражения обеспечивают компактность кода, что в свою очередь минимизирует ошибки.

**Ссылка на Гитхаб: https://github.com/HolyDeadMan**