МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Конструирование ПО»
Тема: Программирование контейнерных классов

Студент гр. 6303 _____ Доброхвалов М. О.

Преподаватель _____ Спицын А. В.

Санкт-Петербург
2019

Оглавление

Цель работы

Изучение среды MS Visual C++, в частности работы в режиме отладки и в режиме профилирования. Реализация иерархии полиморфных классов. Реализация шаблонного контейнерного класса и соответствующего итератора. Работа со стандартной библиотекой шаблонов STL.

Основные теоретические положения

Microsoft Visual C++ (MSVC) — интегрированная среда разработки приложений на языке C++, разработанная корпорацией Microsoft и поставляемая либо как часть комплекта Microsoft Visual Studio.

Библиотека стандартных шаблонов (STL) (англ. Standard Template Library) — набор согласованных обобщённых алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.

Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

Постановка задачи

1. Разработка программ в среде MS Visual C++

1.1. Настройка среды. Выполнение индивидуального задания

1.1.1. Индивидуальное задание: Написать классы для создания графических объектов. Классы должны иметь общий абстрактный базовый класс Shape с чистыми виртуальными функциями.

1.1.2. Необходимо использовать множественное наследование. В классах должны быть предусмотрены виртуальные функции для вывода информации об объектах в поток, а Shape должен иметь дружественный перегруженный оператор <<.

1.1.3. Исходный текст должен быть разделен на три файла .h, .cpp и .cpp с тестовой программой.

1.2. Работа в режиме отладки

1.2.1. Запустить программу и просмотреть ее работу по шагам (Build ->

Start Debug -> Go).

1.2.2. Просмотреть иерархию классов и найти примеры множественного наследования.

1.2.3. Расставить точки прерывания программы (Break Points) и протестировать её работу.

1.2.4. Для выяснения текущих значений переменных, использовать механизм "Watch variable".

1.3. Исследование программы при помощи Profiler

1.3.1. Изучить возможности оптимизации программы в интегрированной среде, в отчете перечислить и объяснить параметры (опции), влияющие на оптимизацию.

1.3.2. Построить несколько вариантов, отличающихся способом оптимизации, проанализировать время работы и объем памяти полученных вариантов. С помощью Profiler определить наиболее долго выполнявшиеся функции. С помощью Profiler определить не исполнявшиеся участки программы.

1.3.3. Изменить текст main так, чтобы выполнялись все участки программы.

2. Применение стандартной библиотеки STL

2.1. *Составить консольные приложения, демонстрирующие основные операции с контейнерами и итераторами STL*

2.1.1. Заполняя 3 контейнера строками из <cstring> или другими элементами, продемонстрировать отличия.

2.1.1.1. последовательностей (vector, list, dequeue);

2.1.1.2. адаптеров последовательностей (stack, queue, priority_queue);

2.1.1.3. ассоциативных контейнеров на базе map.

2.1.2. На примере заполнения одного контейнера-последовательности из предыдущего задания целыми числами, протестировать интерфейсы контейнера и итератора.

2.1.3. Аналогично протестировать ассоциативный контейнер, заполняя его

указателями на разные графические объекты из разд. 1.1. Протестировать алгоритмы-методы и алгоритмы-классы на множестве графических элементов.

2.2. Реализовать новый шаблон контейнера и шаблон итератора для него по индивидуальному заданию

2.2.1. Предусмотреть обработку исключительных ситуаций.

2.2.2. Протестировать контейнер, заполнив его графическими объектами.

2.2.3. В отчете формально описать реализуемую структуру данных и абстракцию итерации, перечислить все отношения между классами, описать интерфейсы классов и особенности реализации.

Выполнение работы

Создадим требуемые классы согласно индивидуальному заданию: базовый абстрактный класс Shape, круг, квадрат, текст и текст в квадрате. Запустим программу в режиме отладки. Результат на рис. 1.

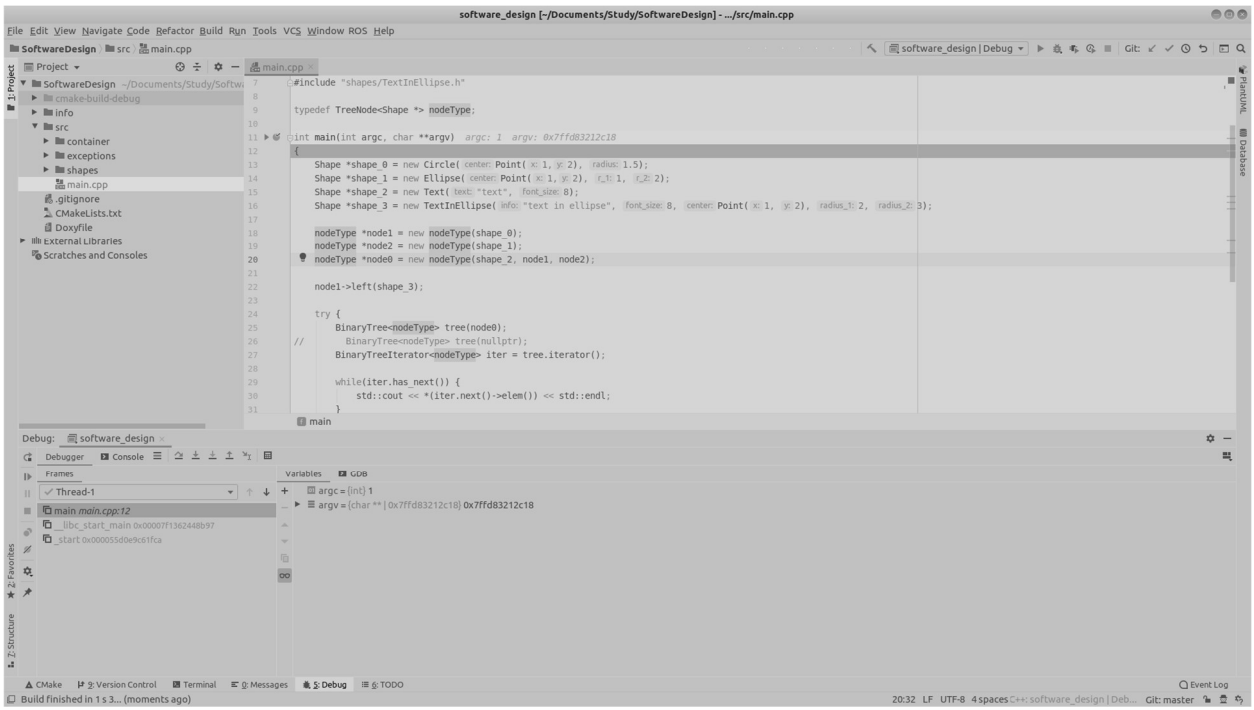

Рис. 1. Запуск программы в режиме отладки

Пример множественного наследования – класс TextInEllipse, который наследуется от Ellipse и Text(Рис 2).

```
#include "Ellipse.h"
#include "Text.h"

class TextInEllipse : public Ellipse, public Text { // center, angle, radius_1.radius_2, text, font_size
public:
    explicit TextInEllipse(std::string info);
    TextInEllipse(std::string info, size_t font_size);
    TextInEllipse(std::string info, float radius_1, float radius_2);
    TextInEllipse(std::string info, Point center, float radius_1, float radius_2);
    TextInEllipse(std::string info, size_t font_size, float radius_1, float radius_2);
    TextInEllipse(std::string info, size_t font_size, Point center, float radius_1, float radius_2);

    void scale(float value) override;

    std::string get_info() const override;

};
```

Рис.ext 2. Класс TextInEllipse

Поставим точку остановы и посмотрим значение переменных(Рис. 3).

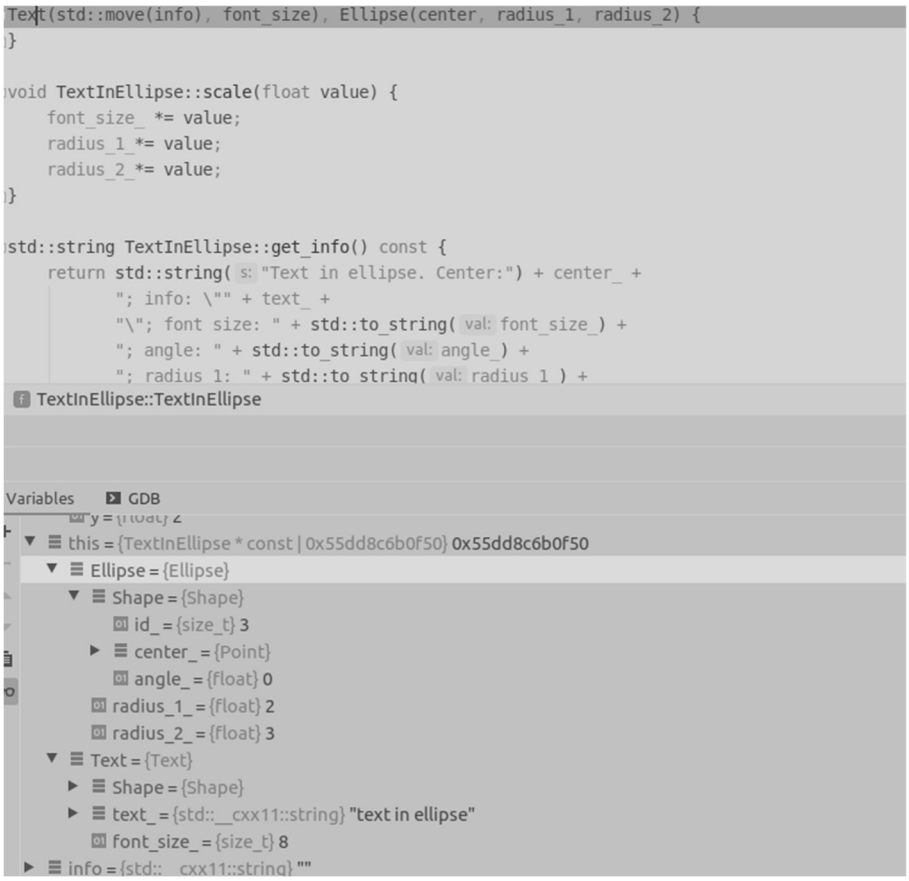

Рис. 3. Остановка на breakpoint

Результат профилирования. Программа gprof(Рис. 4)

Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	53	0.00	0.00	std::remove_reference<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >&>::type&& std::move<std::__cxx11::basic_string<char,
0.00	0.00	0.00	28	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::_
0.00	0.00	0.00	27	0.00	0.00	bool __gnu_cxx::__is_null_pointer<char>(char*)
0.00	0.00	0.00	27	0.00	0.00	void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char*>(char*, char*, std::forward_iterator_tag)
0.00	0.00	0.00	27	0.00	0.00	std::iterator_traits<char*>::difference_type std::__distance<char*>(char*, char*, std::random_access_iterator_tag)
0.00	0.00	0.00	27	0.00	0.00	std::iterator_traits<char*>::iterator_category std::__iterator_category<char*>(char* const&)
0.00	0.00	0.00	27	0.00	0.00	std::iterator_traits<char*>::difference_type std::distance<char*>(char*, char*)
0.00	0.00	0.00	23	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > __gnu_cxx::__to_xstring<std::__cxx11::basic_string<char, std::char_traits<
0.00	0.00	0.00	23	0.00	0.00	void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct<char*>(char*, char*)
0.00	0.00	0.00	23	0.00	0.00	void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_construct_aux<char*>(char*, char*, std::__false_type)
0.00	0.00	0.00	23	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::basic_string<char*, void>(char*, char*, std::allocator<char> const&)
0.00	0.00	0.00	17	0.00	0.00	std::__cxx11::to_string(float)
0.00	0.00	0.00	16	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > std::operator+<char, std::char_traits<char>, std::allocator<char> >(std::_
0.00	0.00	0.00	12	0.00	0.00	TreeNode<Shape*>::left()
0.00	0.00	0.00	12	0.00	0.00	TreeNode<Shape*>*&& std::forward<TreeNode<Shape*>*>(std::remove_reference<TreeNode<Shape*>*>::type&)
0.00	0.00	0.00	10	0.00	0.00	TreeNode<Shape*>::right()
0.00	0.00	0.00	8	0.00	0.00	std::remove_reference<TreeNode<Shape*>*&>::type&& std::move<TreeNode<Shape*>*&>(TreeNode<Shape*>*&)
0.00	0.00	0.00	6	0.00	0.00	std::_Deque_iterator<TreeNode<Shape*>*, TreeNode<Shape*>*&, TreeNode<Shape*>**>::_Deque_iterator(std::_Deque_iterator<TreeNode<Shape*>*, TreeNode<Shape*>*
0.00	0.00	0.00	6	0.00	0.00	std::__cxx11::to_string(unsigned long)
0.00	0.00	0.00	6	0.00	0.00	std::__deque_buf_size(unsigned long)
0.00	0.00	0.00	5	0.00	0.00	BinaryTreeIterator<TreeNode<Shape*> >::has_next()
0.00	0.00	0.00	5	0.00	0.00	std::deque<TreeNode<Shape*>*, std::allocator<TreeNode<Shape*>*> >::empty() const
0.00	0.00	0.00	5	0.00	0.00	std::stack<TreeNode<Shape*>*, std::deque<TreeNode<Shape*>*, std::allocator<TreeNode<Shape*>*> > >::empty() const
0.00	0.00	0.00	5	0.00	0.00	std::deque<TreeNode<Shape*>*, std::allocator<TreeNode<Shape*>*> >::end()
0.00	0.00	0.00	5	0.00	0.00	bool std::operator==<TreeNode<Shape*>*, TreeNode<Shape*>*&, TreeNode<Shape*>**>(std::_Deque_iterator<TreeNode<Shape*>*, TreeNode<Shape*>*&, TreeNode<Shape
0.00	0.00	0.00	4	0.00	0.00	BinaryTreeIterator<TreeNode<Shape*> >::next()

Рис. 4. Результаты профилирования.

Протестируем классы графических объектов(Рис. 5)

```
/home/dmo/Documents/Study/SoftwareDesign/cmake-build-debug/software_design
Text. Center:{0.000000; 0.000000}; info: "text"; font size: 8; angle: 0.000000; id: 2
Circle. Center:{1.000000; 2.000000}; radius: 1.500000; angle: 0.000000; id: 0
Text in ellipse. Center:{0.000000; 0.000000}; info: "text in ellipse"; font size: 8; angle: 0.000000; radius_1: 2.000000; radius_2: 3.000000; id: 3
Ellipse. Center:{1.000000; 2.000000}; radius_1: 1.000000; radius_2: 2.000000; angle: 0.000000; id: 1
```

Рис. 5. Тестирование графических объектов

Сравнение контейнеров (Таблица 1)

Таблица 1.

Тип контейнеров	Последовательностей			Адаптеры последовательностей			
	vector	list	dequeue	stack	queue	priority_queue	map
Доступ с начала	+	+	+	-	+	+	-
Доступ с конца	+	+	+	+	+	-	-
Доступ по индексу	+	-	+	-	-	-	+
Добавление элементов в начало	-	+	+	-	-	+-	-
Добавление элементов в конец	+	+	+	+	+	+-	-
Изменение элементов по индексу	+	-	+	-	-	-	+
Удаление элементов с начала	-	+	+	-	+	+	-
Удаление элементов с конца	+	+	+	+	-	-	-

Выполнена реализация контейнера BinaryTree и соответствующего итератора BinaryTreeIterator. Данный класс представляет бинарное дерево.

Протестируем созданный шаблонный класс и соответствующий итератор (Рис. 6).

```
/home/dmo/Documents/Study/SoftwareDesign/cmake-build-debug/software_design
Text. Center:{0.000000; 0.000000}; info: "text"; font size: 8; angle: 0.000000; id: 2
Circle. Center:{1.000000; 2.000000}; radius: 1.500000; angle: 0.000000; id: 0
Text in ellipse. Center:{0.000000; 0.000000}; info: "text in ellipse"; font size: 8; angle: 0.000000; radius_1: 2.000000; radius_2: 3.000000; id: 3
Ellipse. Center:{1.000000; 2.000000}; radius_1: 1.000000; radius_2: 2.000000; angle: 0.000000; id: 1
```

Рис. 6. Тестирование шаблонного класса и итератора

Исходный код находится в приложениях А, Б и В. Диаграмма классов представлена в приложении Г.

Выводы

Протестированы и изучены контейнерные классы из библиотеки STL. Написаны полиморфные классы графических объектов и шаблонный контейнерный класс – бинарное дерево, и соответствующий итератор.

Список использованных источников

1. Richard L. Halterman. Fundamentals of Programming C++. – Колледждейл, Теннесси, США: School of Computing Southern Adventist University. – 2019. – 785 с.
2. Cppreference. Справка по C++ [Электронный ресурс]. URL: https://ru.cppreference.com/w/ (дата обращения: 14.10.2019).
3. JetBrains. Clion Learning Center [Электронный ресурс]. URL: https://www.jetbrains.com/clion/learning-center/ (дата обращения: 14.10.2019).
4. OpenNet. Профилятор gprof [Электронный ресурс]. URL: https://www.opennet.ru/docs/RUS/gprof/ (дата обращения: 15.10.2019).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД КЛАССОВ ГРАФИЧЕСКИХ ФИГУР

```
#ifndef SOFTWARE_DESIGN_CIRCLE_H
#define SOFTWARE_DESIGN_CIRCLE_H
#include "Shape.h"
class Circle : virtual public Shape {
protected:
    float radius_;
public:
    explicit Circle();
    explicit Circle(float radius);
    explicit Circle(Point center, float radius);
    void scale(float value) override;
    void rotate(float angle) override;
    void rotate(Point axis, float angle) override;
    float &radius() { return radius_;}
    const float &radius() const { return radius_;}
    std::string get_info() const override ;
};
#endif //SOFTWARE_DESIGN_CIRCLE_H
#include "Circle.h"
Circle::Circle() : Shape(), radius_(1){}
Circle::Circle(float radius) : Shape(), radius_(radius){}
Circle::Circle(Point center, float radius = 1) : Shape(center), radius_(radius) {}
void Circle::scale(float value) {
    if (value > 0)
        radius_ *= value;
}
void Circle::rotate(float angle){} // do nothing
void Circle::rotate(Point axis, float angle) {
    center_ = Point{(center_.x - axis.x) * std::cos(angle) - (center_.y - axis.y) * std::sin(angle) + axis.x,
                    (center_.x - axis.x) * std::sin(angle) + (center_.y - axis.y) * std::cos(angle) +
axis.y};
}
std::string Circle::get_info() const {
    return std::string("Circle. Center:") + center_ +
           std::string("; radius: ") + std::to_string(radius_) +
           std::string("; angle: ") + std::to_string(angle_) +
           std::string("; id: ") + std::to_string(id_);
}
#include "Ellipse.h"
Ellipse::Ellipse() : Shape(), radius_1_(1), radius_2_(2){}
Ellipse::Ellipse(float r_1, float r_2) : Shape(), radius_1_(r_1), radius_2_(r_2) {}
Ellipse::Ellipse(Point center, float r_1, float r_2): Shape(center), radius_1_(r_1), radius_2_(r_2) {}
void Ellipse::scale(float value) {
    if (value > 0)
    {
        radius_1_ *= value;
        radius_2_ *= value;
    }
```

```
}
void Ellipse::scale_r1(float value) {
    radius_1_ *= value;
}
void Ellipse::scale_r2(float value) {
    radius_2_ *= value;
}
std::string Ellipse::get_info() const {
    return std::string("Ellipse. Center:") + center_ +
           "; radius_1: " + std::to_string(radius_1_) +
           "; radius_2: " + std::to_string(radius_2_) +
           "; angle: " + std::to_string(angle_) +
           "; id: " + std::to_string(id_);

}
#ifndef SOFTWARE_DESIGN_ELLIPSE_H
#define SOFTWARE_DESIGN_ELLIPSE_H
#include "Shape.h"
class Ellipse : virtual public Shape {
protected:
    float radius_1_;
    float radius_2_;
public:
    explicit Ellipse();
    explicit Ellipse(float r_1, float r_2);
    explicit Ellipse(Point center, float r_1, float r_2);
    void scale(float value) override;
    void scale_r1(float value);
    void scale_r2(float value);

    float &radius_1() { return radius_1_;}
    const float &radius_1() const { return radius_1_;}

    float &radius_2() { return radius_2_;}
    const float &radius_2() const { return radius_2_;}

    std::string get_info() const override ;
};
#endif //SOFTWARE_DESIGN_ELLIPSE_H
#ifndef SOFTWARE_DESIGN_SHAPE_H
#define SOFTWARE_DESIGN_SHAPE_H
#include <iostream>
#include <string>
#include <cmath>
struct Point{
    float x, y;
    Point();
    explicit Point(float x, float y);
    Point &operator=(const Point &other);
    friend std::ostream & operator<<(std::ostream &out, const Point &c);
    friend std::string operator+(const std::basic_string<char>& string, Point point);
```

```
};
class Shape {
protected:
    size_t id_;
    Point center_;
    float angle_;
    Shape();
    explicit Shape(Point center);
public:
    size_t get_id();
    virtual Point &center();
    virtual const Point &center() const;
    virtual float &angle();
    virtual const float &angle() const;
    virtual void scale(float value) = 0;
    virtual void rotate(float angle);
    virtual void rotate(Point axis, float angle);
    virtual void move(Point new_base);
    virtual std::string get_info() const = 0;
    virtual void draw() {}
    virtual void clear() {}
    virtual ~Shape() = default;
    friend std::ostream & operator<<(std::ostream &out, const Shape &c);
};
#endif //SOFTWARE_DESIGN_SHAPE_H
#include "Shape.h"
Point::Point(float x, float y = 0): x(x), y(y){}
Point::Point() : x(0), y(0) {}
Point &Point::operator=(const Point &other) // copy assignment
{
    if (this != &other)
    {
        x = other.x;
        y = other.y;
    }
    return *this;
}
std::ostream &operator<<(std::ostream &out, const Point &c) {
    out << '{' << c.x << "; " << c.y << '}';
    return out;
}
std::string operator+(const std::basic_string<char>& string, Point point) {
    return string + '{' + std::to_string(point.x) + "; " + std::to_string(point.y) + '}';
}
Shape::Shape(Point center) : center_(center), angle_(0){
    static size_t _id = 0;
    id_ = _id;
    ++_id;
}
Shape::Shape() : Shape(Point()){}
Point &Shape::center() { return center_; }
```

```
const Point &Shape::center() const { return center_; }
float &Shape::angle() { return angle_; }
const float &Shape::angle() const { return angle_; }
void Shape::rotate(float angle) {
    angle_ += angle;
}
void Shape::rotate(Point axis, float angle) {
    center_ = Point{(center_.x - axis.x) * std::cos(angle) - (center_.y - axis.y) * std::sin(angle) + axis.x,
                    (center_.x - axis.x) * std::sin(angle) + (center_.y - axis.y) * std::cos(angle) +
axis.y};
    angle_ += angle;
}
void Shape::move(Point new_base) {
    center_ = new_base;
}
std::ostream & operator<<(std::ostream &out, const Shape &c) {
    out << c.get_info();
    return out;
}
#ifndef SOFTWARE_DESIGN_TEXT_H
#define SOFTWARE_DESIGN_TEXT_H
#include <string>
#include "Shape.h"
class Text : virtual public Shape {
protected:
    std::string text_;
    size_t font_size_;
public:
    explicit Text(std::string text);
    explicit Text(std::string text, size_t font_size);
    size_t &font_size();
    const size_t &font_size() const;
    std::string & text();
    const std::string & text() const;
    void scale(float value) override;
    std::string get_info() const override;
};
#endif //SOFTWARE_DESIGN_TEXT_H
#include <utility>
#include "Text.h"
Text::Text(std::string text) : Shape(), text_(std::move(text)), font_size_(8) {}
Text::Text(std::string text, size_t font_size ): Shape(), text_(std::move(text)) {
    font_size_ = font_size ? font_size : 8;
}
size_t &Text::font_size() { return font_size_; }
const size_t &Text::font_size() const { return font_size_; }

void Text::scale(float value) {
    if (value > 0)
        font_size_ *= value;
}
```

```
std::string &Text::text(){
    return text_;
}
const std::string &Text::text() const{
    return text_;
}
std::string Text::get_info() const {
    return std::string("Text. Center:") + center_ +
           "; info: \"" + text_ +
           "\"; font size: " + std::to_string(font_size_) +
           "; angle: " + std::to_string(angle_) +
           std::string("; id: ") + std::to_string(id_);
}
#ifndef SOFTWARE_DESIGN_TEXTINELLIPSE_H
#define SOFTWARE_DESIGN_TEXTINELLIPSE_H
#include "Ellipse.h"
#include "Text.h"
class TextInEllipse : public Ellipse, public Text { // center, angle, radius_1.radius_2, text, font_size
public:
    explicit TextInEllipse(std::string info);
    TextInEllipse(std::string info, size_t font_size);
    TextInEllipse(std::string info, float radius_1, float radius_2);
    TextInEllipse(std::string info, Point center, float radius_1, float radius_2);
    TextInEllipse(std::string info, size_t font_size, float radius_1, float radius_2);
    TextInEllipse(std::string info, size_t font_size, Point center, float radius_1, float radius_2);
    void scale(float value) override;
    std::string get_info() const override;
};
#endif //SOFTWARE_DESIGN_TEXTINELLIPSE_H
#include "TextInEllipse.h"
#include <utility>
TextInEllipse::TextInEllipse(std::string info): Text(std::move(info)){}
TextInEllipse::TextInEllipse(std::string info, size_t font_size) : Text(std::move(info), font_size){}
TextInEllipse::TextInEllipse(std::string info, float radius_1, float radius_2) : Text(std::move(info)),
Ellipse(radius_1, radius_2) {}
TextInEllipse::TextInEllipse(std::string info, Point center, float radius_1, float radius_2) :
Text(std::move(info)), Ellipse(center, radius_1, radius_2){}
TextInEllipse::TextInEllipse(std::string info, size_t font_size, float radius_1, float radius_2) :
    Text(std::move(info), font_size), Ellipse(radius_1, radius_2) {}
TextInEllipse::TextInEllipse(std::string info, size_t font_size, Point center, float radius_1, float
radius_2):
Text(std::move(info), font_size), Ellipse(center, radius_1, radius_2) {}

void TextInEllipse::scale(float value) {
    font_size_ *= value;
    radius_1_*= value;
    radius_2_*= value;
}
std::string TextInEllipse::get_info() const {
    return std::string("Text in ellipse. Center:") + center_ +
           "; info: \"" + text_ +
```

```
            "\"; font size: " + std::to_string(font_size_) +
            "; angle: " + std::to_string(angle_) +
            "; radius_1: " + std::to_string(radius_1_) +
            "; radius_2: " + std::to_string(radius_2_) +
             std::string("; id: ") + std::to_string(id_);
}
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ШАБЛОННОГО КЛАССА И ИТЕРАТОРА ДЛЯ НЕГО

```
#ifndef SOFTWARE_DESIGN_TREENODE_H
#define SOFTWARE_DESIGN_TREENODE_H
#include <utility>
template<typename T>
class TreeNode{
    T elem_;
    TreeNode *left_;
    TreeNode *right_;
public:
    explicit TreeNode(T base);
    TreeNode(T base, TreeNode *elem, bool is_right = false);
    TreeNode(T base, TreeNode* left, TreeNode* right);
    T &elem();
    const T &elem() const;
    void elem(T base);
    TreeNode *left();
    void left(T left);
    void left(TreeNode *left);
    TreeNode *right();
    void right(T right);
    void right(TreeNode *right);
     ~TreeNode();
};
template<typename T>
TreeNode<T>::TreeNode(T base) : elem_(base), left_(nullptr), right_(nullptr){}
template<typename T>
TreeNode<T>::TreeNode(T base, TreeNode* left, TreeNode* right) : elem_(base), left_(left), right_(right){}
template<typename T>
TreeNode<T>::TreeNode(T base, TreeNode *elem, bool is_right) : elem_(base) {
    if (is_right)
    {
        left_ = nullptr;
        right_ = elem;
    }
    else
    {
        left_ = elem;
        right_ = nullptr;
    }
}

template<typename T>
T &TreeNode<T>::elem() { return elem_; }
template<typename T>
const T &TreeNode<T>::elem() const { return elem_; }
template<typename T>
void TreeNode<T>::elem(T base) { elem_ = base; }
template<typename T>
TreeNode<T> *TreeNode<T>::left() { return left_; }
template<typename T>
void TreeNode<T>::left(T left) { left_ = new TreeNode<T>(left); }
template<typename T>
void TreeNode<T>::left(TreeNode *left) { left_ = left;}
template<typename T>
TreeNode<T> *TreeNode<T>::right() { return right_; }
template<typename T>
void TreeNode<T>::right(T right) { right_ = new TreeNode<T>(right); }
template<typename T>
void TreeNode<T>::right(TreeNode *right) {
    right_ = right;
}
template<typename T>
TreeNode<T>::~TreeNode() {
    delete(elem_);
}
#endif //SOFTWARE_DESIGN_TREENODE_H



#ifndef SOFTWARE_DESIGN_BINARYTREE_H
#define SOFTWARE_DESIGN_BINARYTREE_H
#include <iostream>
```

```
#include "TreeNode.h"
#include "../exceptions/EmpyErrorBT.h"
#include "BinaryTreeIterator.h"
template <typename T>
class BinaryTree {
    T *root_;
    void print_klp_(T *node);
public:
    explicit BinaryTree(T *base);
    BinaryTreeIterator<T> iterator(){
        return BinaryTreeIterator<T>(*this);
    }
    T* root() const { return root_; }
    void print_klp();
    void clear(T* node);
    void clear();
    ~BinaryTree();
};
template <typename T>
BinaryTree<T>::BinaryTree(T *base) {
    if (base != nullptr)
        root_ = base;
    else
        throw EmptyErrorBT("pointer is empty");
}
template <typename T>
void BinaryTree<T>::print_klp_(T* node)
{
    std::cout << node->elem() << ' ';
    if (node->left()) {
        print_klp_(node->left());
    }
    if (node->right()) {
        print_klp_(node->right());
    }
}
template <typename T>
void BinaryTree<T>::print_klp()
{
    print_klp_(root_);
}
template <typename T>
void BinaryTree<T>::clear(T* node)
{
    if (node == nullptr)
        return;
    if (node->left()) {
        clear(node->left());
    }
    if (node->right()){
        clear(node->right());
    }
    delete node;
}
template <typename T>
void BinaryTree<T>::clear()
{
    clear(root_);
}

template <typename T>
BinaryTree<T>::~BinaryTree(){
    clear(root_);
    root_ = nullptr;
}
#endif //SOFTWARE_DESIGN_BINARYTREE_H



#ifndef SOFTWARE_DESIGN_BINARYTREEITERATOR_H
#define SOFTWARE_DESIGN_BINARYTREEITERATOR_H

#include <stack>
#include "../exceptions/InvalidIteratorBT.h"
#include "BinaryTree.h"
```

```
template <typename T>
class BinaryTree;

template <typename T>
class BinaryTreeIterator {
    std::stack<T *> stack;
public:
    explicit BinaryTreeIterator(T *root);
    explicit BinaryTreeIterator(const BinaryTree<T> &tree);

    bool has_next();
    T *next();

};

template<typename T>
BinaryTreeIterator<T>::BinaryTreeIterator(T *root) {
    if (root == nullptr)
        throw InvalidIteratorBT("nullptr root");
    stack.push(root);
}

template<typename T>
BinaryTreeIterator<T>::BinaryTreeIterator(const BinaryTree<T> &tree) {
    if (tree.root() == nullptr)
        throw InvalidIteratorBT("nullptr root");
    stack.push(tree.root());
}

template<typename T>
bool BinaryTreeIterator<T>::has_next() {
    return !stack.empty();
}

template<typename T>
T *BinaryTreeIterator<T>::next() {
    T *node = stack.top();
    stack.pop();

    if (node->right())
        stack.push(node->right());

    if (node->left())
        stack.push(node->left());
    return node;
}


#endif //SOFTWARE_DESIGN_BINARYTREEITERATOR_H
```

ПРИЛОЖЕНИЕ В
ИСХОДНЫЙ КОД ТЕСТИРУЮЩЕЙ ПРОГРАММЫ

```
#include "container/BinaryTree.h"
#include "container/BinaryTreeIterator.h"

#include "shapes/Circle.h"
#include "shapes/Ellipse.h"
#include "shapes/Text.h"
#include "shapes/TextInEllipse.h"

typedef TreeNode<Shape *> nodeType;

int main(int argc, char **argv)
{
    Shape *shape_0 = new Circle(Point(1,2), 1.5);
    Shape *shape_1 = new Ellipse(Point(1,2), 1, 2);
    Shape *shape_2 = new Text("text", 8);
    Shape *shape_3 = new TextInEllipse("text in ellipse", 8, Point(1, 2), 2, 3);

    nodeType *node1 = new nodeType(shape_0);
    nodeType *node2 = new nodeType(shape_1);
    nodeType *node0 = new nodeType(shape_2, node1, node2);

    node1->left(shape_3);

    try {
        BinaryTree<nodeType> tree(node0);
//        BinaryTree<nodeType> tree(nullptr);
        BinaryTreeIterator<nodeType> iter = tree.iterator();

        while(iter.has_next()) {
            std::cout << *(iter.next()->elem()) << std::endl;
        }
    }
    catch(EmptyErrorBT &e)
    {
        std::cout << "Exception:" << e.what() << std::endl;
    }
    catch(InvalidIteratorBT &e)
    {
        std::cout << "Exception:" << e.what() << std::endl;
    }
    catch(std::exception &e){
        std::cout << "Exception:" << e.what() << std::endl;
    }

    return 0;
}
```

ПРИЛОЖЕНИЕ Г
ДИАГРАММА КЛАССОВ