Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра: 806 «Вычислительная математика и программирование»
Факультет: «Информационные технологии и прикладная математика»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа №3.

Группа: М8О – 104Б-16
Студент: Чекушкин Денис Игоревич
Преподаватель: Поповкин Александр Викторович
Вариант: №18

Москва
2017

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Закрепление навыков работы с классами.

• Знакомство с умными указателями.

ЗАДАНИЕ

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

Классы должны удовлетворять следующим правилам:

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

• Класс-контейнер должен соджержать объекты используя std:shared_ptr<...>.

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

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

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

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.

Теория:

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).

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.

В новом стандарте появились следующие умные указатели: `unique_ptr`, `shared_ptr` и `weak_ptr`. Все они объявлены в заголовочном файле `<memory>`.

`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).

`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.

`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`. Предположим, есть следующая ситуация (переменные-члены не инкапсулированы для упрощения кода)

Листинг

btree.h

```
#ifndef BTREE_H
#define BTREE_H
#include "btree_item.h"
class Btree
{
public:
      Btree();

      void bstInsert(const std::shared_ptr<Figure>& figure);
      void bstRemove(const std::shared_ptr<Figure>& figure);
      void Insert(const std::shared_ptr<Figure>& figure);
      void print1(const Btree& btree);
```

```
	friend std::ostream& operator << (std::ostream& os, const Btree& Btree);


private:
	std::shared_ptr<BTreeItem> m_root;
};


#endif
btree.cpp
#include "btree.h"
#include "ctype.h"
#include <iostream>
Btree::Btree()
{
m_root=nullptr;
}
void insert_helper(std::shared_ptr<BTreeItem> m_root,const
std::shared_ptr<Figure>& figure)
{
	std::shared_ptr<BTreeItem> node = m_root;


		if(*figure == *(node->getFigure())){ std::cout<<*figure<<"\n";
std::cout<<*(node->getFigure())<<"\n"; std::cout<<"БЫЛО"<<"\n";}
		else{
		if(*figure < *(node->getFigure())) {
			if(node->getLeft()==nullptr)
				{
				std::shared_ptr<BTreeItem>
newl=std::make_shared<BTreeItem>(figure);
				node->setLeft(newl);
				}
			else
				{
				std::shared_ptr<BTreeItem> left = node->getLeft();
				insert_helper(left,figure);
				}
			}
		else if(*figure > *(node->getFigure()))  {
			if(node->getRight()==nullptr)
				{
```

```
                        std::shared_ptr<BTreeItem>
newr=std::make_shared<BTreeItem>(figure);
                        node->setRight(newr);
                        }
                  else
                        {
                        std::shared_ptr<BTreeItem> right = node->getRight();
                        insert_helper(right,figure);
                        }
                  }
 }
 }
 void Btree::bstInsert(const std::shared_ptr<Figure>& figure)
 {
 if(m_root==nullptr)
       {
       m_root = std::make_shared<BTreeItem>(figure);//std::cout<<"INSERT_ROOT";
       }
 else
 {insert_helper(m_root,figure);}
 }
 void remove_helper(std::shared_ptr<BTreeItem> & item, const
std::shared_ptr<Figure>& figure)
 {

     std::shared_ptr<BTreeItem> repl = nullptr, parent = nullptr, tmp = item;
     while (tmp != nullptr && !(*(tmp->getFigure()) == *figure))
     {
         parent = tmp;

         if (*figure < *(tmp->getFigure()))
         {
             tmp = tmp->getLeft();
         }
         else
         {
             tmp = tmp->getRight();
         }
     }
```

```
if (tmp == nullptr)
{
    return;
}
if (tmp->getLeft() != nullptr && tmp->getRight() == nullptr)
{
    if (parent != nullptr)
    {
        if (parent->getLeft() == tmp)
            parent->setLeft(tmp->getLeft());
        else
            parent->setRight(tmp->getRight());
    }
    else
        item = tmp->getLeft();

}
else if (tmp->getLeft() == nullptr && tmp->getRight() != nullptr)
{
    if (parent != nullptr)
    {
        if (parent->getLeft() == tmp)
            parent->setLeft(tmp->getLeft());
        else
            parent->setRight(tmp->getRight());
    }
    else
        item = tmp->getRight();

}
else if (tmp->getLeft() != nullptr && tmp->getRight() != nullptr)
{
    repl = tmp->getRight();

    if (repl->getLeft() == nullptr) {
        tmp->setRight(repl->getRight());
    }
```

```
        else
        {
            while (repl->getLeft() != nullptr)
            {
                parent = repl;
                repl = repl->getLeft();
            }

            parent->setLeft(repl->getRight());
        }
        *(tmp->getFigure()) = *(repl->getFigure());
    }
    else
    {
        if (parent != nullptr)
        {
            if (parent->getLeft() == tmp)
                parent->setLeft(nullptr);
            else
                parent->setRight(nullptr);
        }
        else
            item = nullptr;
    }
    return;
}
void Btree::bstRemove(const std::shared_ptr<Figure>& figure)
{
    remove_helper(m_root, figure);
}
void print2(const std::shared_ptr<BTreeItem> &node)
{
    if(node == nullptr) return;
    print2(node->getLeft());
    std::cout << *(node->getFigure())<< std::endl;
    print2(node->getRight());
}
```

```
        std::ostream& operator<<(std::ostream & os, const Btree &btree)
        {
        std::shared_ptr<BTreeItem> item=btree.m_root;
            print2(item);
        return os;
        }
```

btree_item.h

```
        #ifndef BTREE_ITEM_H
        #define BTREE_ITEM_H

        #include <memory>
        #include "figure.h"

        class BTreeItem
        {
        public:
              BTreeItem(const std::shared_ptr<Figure>& figure);
              friend std::ostream& operator << (std::ostream& os, const BTreeItem& obj);
              bool operator == (const BTreeItem& other) const;

              void setLeft(std::shared_ptr<BTreeItem> left);
              void setRight(std::shared_ptr<BTreeItem> right);
              std::shared_ptr<BTreeItem> getLeft();
              std::shared_ptr<BTreeItem> getRight();
              std::shared_ptr<Figure> getFigure() const;

        private:
        std::shared_ptr<Figure> m_figure;
        std::shared_ptr<BTreeItem> m_left;
        std::shared_ptr<BTreeItem> m_right;
        };
        #endif
```

btree_item.cpp

```
        #include "btree_item.h"
        #include <iostream>
        BTreeItem::BTreeItem(const std::shared_ptr<Figure>& figure)
        {
```

```
 m_left = nullptr;
 m_right = nullptr;
 m_figure = figure;
}
bool BTreeItem::operator == (const BTreeItem& other) const
{
      return getFigure() == other.getFigure();
}
void BTreeItem::setLeft(std::shared_ptr<BTreeItem> left)
{
m_left=left;
}
void BTreeItem::setRight(std::shared_ptr<BTreeItem> right)
{
m_right=right;
}
std::shared_ptr<BTreeItem> BTreeItem::getLeft()
{
return m_left;
}
std::shared_ptr<BTreeItem> BTreeItem::getRight()
{
return m_right;
}
std::shared_ptr<Figure> BTreeItem::getFigure() const
{
return m_figure;
}
std::ostream& operator << (std::ostream& os, const BTreeItem& obj) {
    os << "[" << *obj.m_figure << "]" << std::endl;
    return os;
}
```

square.h

```
#ifndef SQUARE_H
#define SQUARE_H
#include <iostream>
#include "figure.h"
```

```
class Square : public Figure
{
public:
	Square();
	Square(std::istream& is);
	Square(size_t i);
	void print() const override;
	double area() const override;
	Square& operator = (const Square& other);
	bool operator == (const Square& other) const;
	bool operator < (const Square& rhs) const;
	bool operator > (const Square& rhs) const;

	friend std::ostream& operator << (std::ostream& os, const Square& square);
	friend std::istream& operator >> (std::istream& is, Square& square);
private:
	double m_side;
};

#endif
```

sqaure.cpp

```
#include "square.h"
Square::Square()
{
	m_side = 0.0;
}
Square::Square(std::istream& is)
{
	std::cout << "================" << std::endl;
	std::cout << "Enter side: ";
	is >> m_side;
}
Square::Square(size_t i) : m_side(i) //!!
{
    std::cout << "Square passed to function. Side: " << m_side << std::endl;
}
void Square::print() const
```

```
{
	std::cout << "================" << std::endl;
	std::cout << "Figure type: square" << std::endl;
	std::cout << "Side size: " << m_side << std::endl;
}
double Square::area() const
{
	return m_side * m_side;
}
Square& Square::operator = (const Square& other)
{
	if (&other == this)
		return *this;
	m_side = other.m_side;
	return *this;
}
bool Square::operator < (const Square& rhs) const
{
  return (this->area() < rhs.area());
}
bool Square::operator > (const Square& rhs) const
{
  return (this->area() > rhs.area());
}
bool Square::operator == (const Square& other) const
{
	return m_side == other.m_side;
}
bool Figure::operator > (const Figure& rhs) const
	{
	  return (this->area() > rhs.area());
	}
bool Figure::operator < (const Figure& rhs) const
	{
	  return (this->area() < rhs.area());
	}
bool Figure::operator == (const Figure& rhs) const
	{
```

```
        return area() == rhs.area();
    }
std::ostream& operator << (std::ostream& os, const Square& square)
{
    os << "================" << std::endl;
    os << "Figure type: square" << std::endl;
    os << "Side size: " << square.m_side << std::endl;

    return os;
}
std::istream& operator >> (std::istream& is, Square& square)
{
    std::cout << "================" << std::endl;
    std::cout << "Enter side: ";
    is >> square.m_side;

    return is;
}
std::ostream& operator << (std::ostream& os, const Figure& figure)
{
    figure.print();
    return os;
}
```

main.cpp

```
#include "btree.h"
#include "square.h"
#include "rectangle.h"
#include "trapezoid.h"
#include <iostream>
int main()
{
    unsigned int action;
    Btree b;
    while (true)
    {
        std::cout << "================" << std::endl;
        std::cout << "Menu:" << std::endl;
        std::cout << "1) Add figure" << std::endl;
```

```
		std::cout << "2) Delete figure" << std::endl;
		std::cout << "3) Print" << std::endl;
		std::cout << "0) Quit" << std::endl;
		std::cin >> action;
		if (action == 0)
			break;
		if (action > 3)
		{
			std::cout << "Error: invalid action" << std::endl;
			continue;
		}
		switch (action)
		{
			case 1:
			{
				unsigned int figureType;
				std::cout << "================" << std::endl;
				std::cout << "1) Square" << std::endl;
				std::cout << "2) Rectangle" << std::endl;
				std::cout << "3) Trapezoid" << std::endl;
				std::cout << "0) Quit" << std::endl;
				std::cin >> figureType;
				if (figureType > 0)
				{
					if (figureType > 3)
					{
						std::cout << "Error: invalid figure type"
<< std::endl;
						continue;
					}
					switch (figureType)
					{
						case 1:
						{
	b.bstInsert(std::make_shared<Square>(std::cin));
							break;
						}
```

```
                                        case 2:
                                        {
b.bstInsert(std::make_shared<Rectangle>(std::cin));
                                              break;
                                        }
                                        case 3:
                                        {
b.bstInsert(std::make_shared<Trapezoid>(std::cin));
                                              break;
                                        }
                                  }
                            }
                            break;
 }
                     case 2:
                     {
                            unsigned int figureType;
                            std::cout << "================" << std::endl;
                            std::cout << "1) Square" << std::endl;
                            std::cout << "2) Rectangle" << std::endl;
                            std::cout << "3) Trapezoid" << std::endl;
                            std::cout << "0) Quit" << std::endl;
                            std::cin >> figureType;
                            if (figureType > 0)
                            {
                                  if (figureType > 3)
                                  {
                                        std::cout << "Error: invalid figure type"
<< std::endl;
                                        continue;
                                  }
                                  switch (figureType)
                                  {
                                        case 1:
                                        {
```

```
b.bstRemove(std::make_shared<Square>(std::cin));

                                                        break;
                                                }
                                                case 2:
                                                {

b.bstRemove(std::make_shared<Rectangle>(std::cin));

                                                        break;
                                                }
                                                case 3:
                                                {

b.bstRemove(std::make_shared<Trapezoid>(std::cin));

                                                        break;
                                                }
                                        }
                                }
                                break;
                        }
                        case 3:
                        {
                                std::cout<<b;
                                break;
                }
        }
}
return 0;
}
```

Выводы: в данной лабораторной работе я закрепил навыки пограммирования классов на языке C++, научился работать с умными указателями. Добавил фигуры: прямоугольник, трапецию. Добвил умные указатели в структуру, в фигуры.

https://github.com/israelcode/oop/tree/master/sem2/lab3