Московский Авиационный Институт
(Национальный Исследовательский Университет)

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

Лабораторная работа №2.

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.
• Создание простых динамических структур данных.
• Работа с объектами, передаваемыми «по значению».

ЗАДАНИЕ

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

Классы должны удовлетворять следующим правилам:

• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
• Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<).
Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>).
Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
• Классы фигур должны иметь операторы копирования (=).
• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
• Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).
• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер
• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).
• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).
• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

• Стандартные контейнеры std.
• Шаблоны (template).
• Различные варианты умных указателей (shared_ptr, weak_ptr).
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

Теория

Классы в C++ — 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.

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

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:

- она не имеет имени;
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Листинг

sqaure.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;
}
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;
    }
```

btree_item.h

```
#ifndef BTREE_ITEM_H
#define BTREE_ITEM_H
#include "square.h"
class BTreeItem
{
public:
BTreeItem(const Square& square);
friend std::ostream& operator << (std::ostream& os, const BTreeItem& obj);
void setLeft(BTreeItem* left);
void setRight(BTreeItem* right);
BTreeItem* getLeft();
BTreeItem* getRight();
Square getSquare() const;
virtual ~BTreeItem ();

private:
Square m_square;
BTreeItem* m_left;
BTreeItem* m_right;
};
#endif
```

btreeitem.cpp

```
#include "btree_item.h"
#include <iostream>
BTreeItem::BTreeItem(const Square& square) {
 m_left = nullptr;
 m_right = nullptr;
 m_square = square;
}
```

```
void BTreeItem::setLeft(BTreeItem* left) {
m_left=left;
}
void BTreeItem::setRight(BTreeItem* right) {
m_right=right;
}
BTreeItem* BTreeItem::getLeft() {
return m_left;
}
BTreeItem* BTreeItem::getRight() {
return m_right;
}
Square BTreeItem::getSquare() const {
return m_square;
}
BTreeItem::~BTreeItem() {
delete m_left;
delete m_right;
}
std::ostream& operator << (std::ostream& os, const BTreeItem& obj) {
    os << "[" << obj.m_square << "]" << std::endl;
    return os;
}
```

btree.h

```
#ifndef BTREE_H
#define BTREE_H
#include "btree_item.h"
class Btree
{
public:
        Btree();
        virtual ~Btree();
        void bstInsert(const Square& square);
        void bstRemove(const Square& square);
        friend std::ostream& operator << (std::ostream& os, const Btree& Btree);
```

```
private:
        BTreeItem* m_root;
        };
        #endif
```

btree.cpp

```
#include "btree.h"
#include "ctype.h"
Btree::Btree()
{
m_root=nullptr;
}
void destroy_tree(BTreeItem *m_root)
{
 if(m_root!=nullptr)
 {
   destroy_tree(m_root->getLeft());
   destroy_tree(m_root->getRight());
   delete m_root;
 }
}
Btree::~Btree() {
 destroy_tree(m_root);
}
void insert_helper(BTreeItem **ptr,const Square& value)
{

        if (value<(*ptr)->getSquare())
        {
                if ((*ptr)->getLeft())
                {
                BTreeItem* left = (*ptr)->getLeft();
                insert_helper(&left,value);
                }
                else
```

```
                {
                BTreeItem* newl= new BTreeItem(value);
                (*ptr)->setLeft(newl);
                }
        }
        else if (value>(*ptr)->getSquare())
        {
                if((*ptr)->getRight())
                {
                BTreeItem* right = (*ptr)->getRight();
                insert_helper(&right,value);
                }
                else
                {
                BTreeItem* newr= new BTreeItem(value);
                (*ptr)->setRight(newr);
                }
        }
        else if (value==(*ptr)->getSquare())
        {std::cout<<"Было"<<std::endl;}
}

void Btree::bstInsert(const Square& square)
{
if(m_root==nullptr)
{m_root= new BTreeItem(square);}
else {insert_helper(&m_root,square);}
}
std::ostream& operator<<(std::ostream& os, const Btree& btree)
{
        BTreeItem *item=btree.m_root;
        if (item==nullptr) {
        os << "================" << std::endl;
        os << "Tree is empty" << std::endl;
        }
```

```
	else
	{
		BTreeItem *left = item->getLeft();
			if (left) {
			os<<*(left);
			}
		os<<(item->getSquare());
		BTreeItem *right = item->getRight();
			if (right) {
			os<<*(right);
			}
	}
	return os;
}
void remove_helper(BTreeItem **item, const Square& square)
{
BTreeItem *repl = nullptr, *parent = nullptr, *tmp = *item;
while (tmp != nullptr && !(tmp->getSquare() == square))
	{
		parent = tmp;
		if (square < tmp->getSquare())
			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->getRight();
        free(tmp);
        tmp = nullptr;
}
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();
        free(tmp);
        tmp = nullptr;
}
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->getSquare() = repl->getSquare();
                free(repl);
                repl = nullptr;
        }
        else
        {
                if (parent != nullptr)
                {
                        if (parent->getLeft() == tmp)
                                parent->setLeft(nullptr);
                        else
                                parent->setRight(nullptr);
                }
                else
                        *item = nullptr;
                free(tmp);
                tmp = nullptr;
        }
        return ;
}
void Btree::bstRemove(const Square& square) {
        remove_helper(&m_root, square);
}
```

main.cpp

```
#include "btree.h"
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: {
                    b.bstInsert(std::cin);
                    break;
                }
                case 2: {
                    Square square(std::cin);
                    b.bstRemove(square);
                    break;
                }
                case 3: {
                    std::cout << b;
                    break;
                }
            }
        }
        return 0;
    }
```

Выводы: в данной лабораторной работе я закрепил навыки пограммирования классов на языке C++, закрепил навыки создания динамических структур.

https://github.com/israelcode/oop/tree/master/sem2/lab2