Московский Авиационный Институт
(Национальный Исследовательский Университет)

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

Лабораторная работа №4.

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

Москва
2017

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Знакомство с шаблонами классов.

• Построение шаблонов динамических структур данных.

ЗАДАНИЕ

Необходимо спроектировать изапрограммировать 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).

Теория

Шабло́ны — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

В C++ возможно создание шаблонов функций и классов.

Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем, ссылка).

Любой шаблон начинается со слова template, будь то шаблон функции или шаблон класса. После ключевого слова template идут угловые скобки — < >, в которых перечисляется список параметров шаблона. Каждому параметру должно предшествовать зарезервированное слово class или typename. Отсутствие этих ключевых слов будет расцениваться компилятором как синтаксическая ошибка.

Ключевое слово typename говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как: int, double,float, char и т. д. А ключевое слово class сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться пользовательские типы данных, то есть классы. Но не в коем случае не путайте параметр шаблона и шаблон класса.

Листинг

btree.h

```
#ifndef BTREE_H
#define BTREE_H
#include <iostream>
#include "btree_item.h"
template <class T>
class Btree
{
public:
 Btree();
 void bstInsert(const std::shared_ptr<T>& figure);
 void bstRemove(const std::shared_ptr<T>& figure);
 void Insert(const std::shared_ptr<T>& figure);
 template <class B>
 friend std::ostream& operator << (std::ostream& os, const Btree<B>& Btree);
 template <class B>
 void print1(const Btree<B>& btree);
private:
 std::shared_ptr<BTreeItem<T>> m_root;
};
#include "btree.cpp"
#endif
```

btree.cpp

```
template <class T>
Btree<T>::Btree()
{
m_root=nullptr;
}
template <class T>
void insert_helper(std::shared_ptr<BTreeItem<T>> m_root,const std::shared_ptr<T>&
figure)
{
```

```
       std::shared_ptr<BTreeItem<T>> 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<T>>
newl=std::make_shared<BTreeItem<T>>(figure);
                            node->setLeft(newl);
                            }
                     else
                            {
                            std::shared_ptr<BTreeItem<T>> left = node->getLeft();
                            insert_helper(left,figure);
                            }
                     }
              else if(*figure > *(node->getFigure()))  {
                     if(node->getRight()==nullptr)
                            {
                            std::shared_ptr<BTreeItem<T>>
newr=std::make_shared<BTreeItem<T>>(figure);
                            node->setRight(newr);
                            }
                     else
                            {
                            std::shared_ptr<BTreeItem<T>> right = node->getRight();
                            insert_helper(right,figure);
                            }
                     }
      }
      }
      template <class T>
      void Btree<T>::bstInsert(const std::shared_ptr<T>& figure)
      {
      if(m_root==nullptr)
       {
       m_root = std::make_shared<BTreeItem<T>>(figure);
```

```
        }
      else
      {insert_helper(m_root,figure);
      }
      }
      template <class T>
      void remove_helper(std::shared_ptr<BTreeItem<T>> & item, const std::shared_ptr<T>&
figure)
      {
          std::shared_ptr<BTreeItem<T>> 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;
    }
    template <class T>
    void Btree<T>::bstRemove(const std::shared_ptr<T>& figure)
    {
        remove_helper(m_root, figure);
    }
    template <class T>
    void print2(const std::shared_ptr<BTreeItem<T>> &node)
    {
        if(node == nullptr) return;
        print2(node->getLeft());
        std::cout << *(node->getFigure())<< std::endl;
        print2(node->getRight());
    }
    template <class B>
    std::ostream& operator<<(std::ostream& os, const Btree<B> &btree)
    {
    std::shared_ptr<BTreeItem<B>> item=btree.m_root;
    print2(item);
    return os;
    }
```

btree_item.h

```
    #ifndef BTREE_ITEM_H
    #define BTREE_ITEM_H
    #include <memory>
```

```
template <class T>
class BTreeItem
{
public:
 BTreeItem(const std::shared_ptr<T>& figure);
 template <class B>
 friend std::ostream& operator << (std::ostream& os, const BTreeItem<B>& obj);
 bool operator == (const BTreeItem<T>& other) const;
 void setLeft(std::shared_ptr<BTreeItem<T>> left);
 void setRight(std::shared_ptr<BTreeItem<T>> right);
 std::shared_ptr<BTreeItem<T>> getLeft();
 std::shared_ptr<BTreeItem<T>> getRight();
 std::shared_ptr<T> getFigure() const;

private:
std::shared_ptr<T> m_figure;
std::shared_ptr<BTreeItem<T>> m_left;
std::shared_ptr<BTreeItem<T>> m_right;
};
#include "btree_item.cpp"
#endif
```

btree_item.cpp

```
template <class T>
BTreeItem<T>::BTreeItem(const std::shared_ptr<T>& figure)
{
 m_left = nullptr;
 m_right = nullptr;
 m_figure = figure;
}
template <class T>
bool BTreeItem<T>::operator == (const BTreeItem<T>& other) const
{
 return getFigure() == other.getFigure();
}

template <class T>
```

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

Выводы: в данной лабораторной работе я получил навыки построения шаблонов динамических структур данных. Добавил фигуры: прямоугольник, трапецию. Добвил шаблоны в классы.

https://github.com/israelcode/oop/tree/master/sem2/lab4