**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 6**

Тема: Шаблоны классов в языке языке С++

Студент: Ваньков Д. А.

Группа: 80-207Б-17

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2018

Вариант задания

Вариант 22: дерево с фигурами: пятиугольник, шестиугольник,

восьмиугольник.

Контейнер 2-го уровня: стек.

Постановка задачи

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. Аллокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-ого уровня, согласно варианту задания). Для вызова аллокатора должны быть переопределены операторы new и delete у классов-фигур.

Программа должна позволять:

* Вводить произвольное количество фигур и добавлять их в контейнер.
* Распечатывать содержимое контейнера.
* Удалять фигуры из контейнера.

Описание реализации программы

Все фигуры являются подклассами виртуального класса figure, в котором описаны основные методы для фигур: *Square* — нахождение площади данной фигуры, *Print —* вывод параметров фигуры и деструктор объекта. В функции *main* стоит switch Case, на котором построен интерфейс программы. Пользователь вводит некоторое число, чтобы запустить необходимую ему функцию (подробнее в следующем пункте). Из switch Case вставка и удаление элемента, содержащего в себе фигуру, выполняется вывод всех элементов очереди и завершение приложения. Для каждой фигуры реализованы конструкторы, для вызова без параметров, для вызова с параметрами в качестве необходимого количества чисел с плавающей точкой, для вызова с потоком ввода в качестве параметра и для вызова со ссылкой на объект того же класса (конструктор копирования). В конструкторах была написана проверка введенных данных (стороны фигуры не могут быть отрицательными). Естественно в классе каждой фигуры реализованы процедуры описанные в *figure*: *Square* и *Print* и деструктор (который не отличается от стандартного деструктора из-за того, что в данных классах не использовались динамические типы)*.*  При вводе параметров конкретной фигуры, производится проверка на ее существование, в случае неверно введенных параметров – выведется сообщение об этом, и к фигуре применяются нулевые значения. Для элементов стека реализовано два конструктора: копирования и обычный, сеттер и геттер элемента стека, геттер фигуры, оператор вывода параметров элемента стека и деструктор. Для класса стек были реализованы два конструктора, функции добавления и удаления элемента, функции получения первого и последнего элемента, оператор вывода всех элементов очереди. И наконец для удобства запуска и компиляции программы был написан *Makefile*.

Описание построения дерева

Каждому элементу присваивается свой ключ, по которому он добавляется как сын предка, ключ которого мы вводим. Если дерево пустое первый элемент встает на место корня. Если при вводе, был не найден предок с определенным ключом, элемент встает как брат для последнего сына.

Для удаления элемента из дерева нужно ввести номер его ключа. Если удаляется элемент, у которого есть дети, удаляются также все дети этого элемента.

Руководство по эксплуатации

Для того, чтобы скомпилировать программу, запустите *make* в директории с исходным кодом. Чтобы запустить программу необходимо ввести *./a.out* в командной строке. При запуске программа выведет сообщение о том, какие команды необходимо ввести для вызова конкретных функций:

1. Еще раз вывести руководство
2. Добавить пятиугольник
3. Добавить шестиугольник
4. Добавить восьмиугольник
5. Удалить фигуру из дерева и вывести её параметры
6. Вывести все дерево

Для завершения программы нужно ввести любой другой символ.

Краткий обзор модулей программы:

|  |  |
| --- | --- |
| Figure.h | Модуль отвечающий за описание общего родительского класса фигур |
| Octagon.h | Заголовочный файл, описывающий класс “Восьмиугольник” |
| Octagon.cpp | Файл, отвечающий за операции над восьмиугольником, содержащий все необходимые функции и переменные, для этого класса |
| Hexagon.h | Заголовочный файл, описывающий класс “Шестиугольник” |
| Hexagon.cpp | Файл, отвечающий за операции над шестиугольником, содержащий все необходимые функции и переменные, для этого класса |
| Pentagon.h | Заголовочный файл, описывающий класс “Пятиугольник” |
| Pentagon.cpp | Файл, отвечающий за операции над пятиугольником, содержащий все необходимые функции и переменные, для этого класса |
| Tree.h | Заголовочный файл, описывающий контейнер “дерево” |
| Tree.cpp | Файл, отвечающий за операции над деревом, такие как вставка, удаление и т.д. |
| Item.h | Заголовочный файла Item.cpp |
| Item.cpp | Файл, отвечающий за связь дерева с фигурами, которыми он будет заполняться |
| Iterator.h | Заголовочный файл, описывающий структуру итераторов |
| TStackItem.h | Заголовочный файла STackItem.cpp |
| TStackItem.cpp | Файл, описывающий функции элемента стека |
| TStack.h | Заголовочный файла STack.cpp |
| TStack.cpp | Файл, отвечающий за описание функций стека, в качестве контейнера 2-го уровня |
| TAllocatationBlock.h | Заголовочный файла TAllocatationBlock.cpp |
| TAllocatationBlock.cpp | Файл, отвечающий за реализацию аллокатора памяти |
| main.cpp | Модуль, отвечающий за “лицо” программы, содержащий интерфейс, позволяющий работать с написанным в других файлах кодом |

Пример выполнения программы

C:\Users\dvankov\CLionProjects\OOP\_lab5\cmake-build-debug\OOP\_lab4.exe

Press 1 to get help

Press 2 to add figure in tree

Press 3 to get a number of nodes on level

Press 4 to print tree

Press 5 to delete figure from tree

Press 6 to get demo iterator

Press 0 to exit

1

Press 1 to get help

Press 2 to add figure in tree

Press 3 to get a number of nodes on level

Press 4 to print tree

Press 5 to delete figure from tree

Press 6 to get demo iterator

Press 0 to exit

2

Choose the figure:

1 - Pentagon

2 - Hexagon

3 - Octagon

3

Insert side of Octagon:

4

Allocated :32bytes

Tree item: created

2

Choose the figure:

1 - Pentagon

2 - Hexagon

3 - Octagon

1

Insert side of Pentagon:

5

Allocated :32bytes

Tree item: created

4

Printed tree:

[Octagon with side = 4]

[Pentagon with side a = 5]

2

Choose the figure:

1 - Pentagon

2 - Hexagon

3 - Octagon

3

Insert side of Octagon:

6

Allocated :32bytes

Tree item: created

4

Printed tree:

[Octagon with side = 4]

[Pentagon with side a = 5]

[Octagon with side = 6]

1

Press 1 to get help

Press 2 to add figure in tree

Press 3 to get a number of nodes on level

Press 4 to print tree

Press 5 to delete figure from tree

Press 6 to get demo iterator

Press 0 to exit

6

Octagon with side = 4

Pentagon with side a = 5

Octagon with side = 6

**main.cpp**

//Ваньков Денис М80-207Б-17

//Вариант 22(дерево, пятиугольник, шестиугольник, восьмиугольник)

#include <cstdlib>

#include <iostream>

#include <memory>

#include <cstdint>

#include "Pentagon.h"

#include "Hexagon.h"

#include "Octagon.h"

#include "Item.h"

#include "Tree.h"

#include "Figure.h"

void help(){

std::cout << "Press 1 to get help" << std::endl;

std::cout << "Press 2 to add figure in tree" << std::endl;

std::cout << "Press 3 to get a number of nodes on level" << std::endl;

std::cout << "Press 4 to print tree" << std::endl;

std::cout << "Press 5 to delete figure from tree" << std::endl;

std::cout << "Press 0 to exit" << std::endl;

}

int main(int argc, const char \* argv[]) {

size\_t act;

TTree<Figure> s;

help();

while (true) {

std::cin >> act;

if (act == 0) break;

if (act > 5) {

std::cout << "Error: enter another parameter (0-5)\n";

continue;

}

switch (act) {

case 1: {

help();

break;

}

case 2: {

size\_t type;

std::cout << "Choose the figure: \n";

std::cout << "1 - Pentagon\n";

std::cout << "2 - Hexagon\n";

std::cout << "3 - Octagon\n";

std::cin >> type;

if (type > 0) {

if (type > 3) {

std::cout << "Error: enter another parameter (0-3)\n";

continue;

}

switch (type) {

case 1: {

s.Push(std::shared\_ptr<TTreeItem<Figure>> (new TTreeItem<Figure> (std::shared\_ptr <Pentagon>(new Pentagon(std::cin)))));

break;

}

case 2: {

s.Push(std::shared\_ptr<TTreeItem<Figure>> (new TTreeItem<Figure> (std::shared\_ptr <Hexagon>(new Hexagon(std::cin)))));

break;

}

case 3: {

s.Push(std::shared\_ptr<TTreeItem<Figure>> (new TTreeItem<Figure> (std::shared\_ptr <Octagon>(new Octagon(std::cin)))));

break;

}

}

}

break;

}

case 3: {

int number = 0;

std::cout << "Enter level to get a number of nodes" << std::endl;

std::cin >> number;

std::cout << "On level " << number << " nodes: " << s.GetNodesOnLevel(number) << std::endl;

break;

}

case 4: {

std::cout << s << "\n";

break;

}

case 5: {

if (s.empty())

std::cout << "Tree is empty" << std::endl;

else {

std::cout << "Enter side of figure to delete it: " << std::endl;

size\_t value;

std::cin >> value;

s.Pop(value);

}

break;

}

case 6: {

if (s.empty())

std::cout << "Tree is empty" << std::endl;

else {

for (auto i : s) {

i->Print();

std::cout << std::endl;

}

}

break;

}

default: {

std::cerr << "Error: incorrect data\n";

break;

}

}

}

return 0;

}

**Tree.cpp**

#include <iostream>

#include "Tree.h"

#include "Item.h"

#include "Figure.h"

template <class T>

TTree<T>::TTree() {

this->root = nullptr;

}

template <class T>

void TreeDel(std::shared\_ptr<TTreeItem<T>> item) { // Деструктор для дерева

if(item) {

TreeDel(item->GetBrother());

TreeDel(item->GetSon());

item.reset();

}

}

template <class T>

TTree<T>::~TTree() {

TreeDel(root);

std::cout << "Tree deleted" << std::endl;

}

template <class T>

int TTree<T>::GetNodesOnLevel(std::shared\_ptr<TTreeItem<T>> item, int level) { //Функция показывающаю, сколько элементов на определенном уровне

if (item == nullptr) return 0;

if (level <= 0) return 0;

return GetNodesOnLevel(item->GetSon(), level - 1) + (level == 1) + GetNodesOnLevel(item->GetBrother(), level);

}

template <class T>

bool TTree<T>::empty() {

return root == nullptr;

}

template <class T>

std::shared\_ptr<TTreeItem<T>> TTree<T>::Push(std::shared\_ptr<TTreeItem<T>> item)

{

if (this->empty()) {

this->root = item;

return this->root;

} else {

if (root->GetSon()) {

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

while (tmp->GetBrother())

tmp = tmp->GetBrother();

tmp->SetBrother(item);

return tmp->GetBrother();

}

else {

root->SetSon(item);

return root->GetSon();

}

}

}

template <class T>

std::shared\_ptr<TTreeItem<T>> tree\_find\_parent(std::shared\_ptr<TTreeItem<T>> item, int value) { //функция по поиску предка по значению

if (item == nullptr) {

return nullptr;

}

if(item->GetFigure()->GetSide() == value) {

return item;

}

std::shared\_ptr<TTreeItem<T>> parent\_node = tree\_find\_parent(item->GetSon(), value);

if (parent\_node == nullptr) {

parent\_node = tree\_find\_parent(item->GetBrother(), value);

}

return parent\_node;

}

template <class T>

int tree\_find\_parent1(std::shared\_ptr<TTreeItem<T>> item, int value) { // Функция поиска предка по указателю на значение

int idx = 0;

if (item->GetFigure()->GetSide() == value) {

return 1;

}

if (item->GetSon() != nullptr){

idx = tree\_find\_parent1(item->GetSon() , value);

}

if (item->GetBrother() != nullptr){

idx = tree\_find\_parent1(item->GetBrother(), value);

}

if (idx == 1) {

item->GetBrother().reset();

item->GetBrother() = nullptr;

idx = 0;

}

return 0;

}

template <class T>

void split(std::shared\_ptr<TTreeItem<T>> item) { //Функция для переприсваивания брата (перескок)

std::shared\_ptr<TTreeItem<T>> tmp = item->GetBrother();

item->GetSon() = tmp->GetSon();

item->GetFigure() = tmp->GetFigure();

item->GetBrother() = tmp->GetBrother();

tmp.reset();

tmp = nullptr;

}

template <class T>

TTreeItem<T> \*destroy\_Tree(std::shared\_ptr<TTreeItem<T>> pointer) {

if(pointer == nullptr) {

return nullptr;

}

if (pointer->GetSon() != nullptr) {

destroy\_Tree(pointer->GetSon());

}

if (pointer->GetBrother() != nullptr) {

destroy\_Tree(pointer->GetBrother());

}

if (pointer->GetSon() == nullptr && pointer->GetBrother() == nullptr) {

pointer.reset();

pointer = nullptr;

}

return nullptr;

}

template <class T>

void TTree<T>::Pop(size\_t value) // Вызов функции удаления

{

if (root->GetFigure()->GetSide() == value) {

root = nullptr;

}

else {

Pop(root, value);

}

}

template <class T>

void TTree<T>::Pop(std::shared\_ptr<TTreeItem<T>> item, size\_t value) // Удаление элемента из дерева

{

if (item->GetSon()) {

if (item->GetSon()->GetFigure()->GetSide() == value) {

std::shared\_ptr<TTreeItem<T>> ptr = item->GetSon();

item->SetSon(item->GetSon()->GetBrother());

ptr->SetBrother(nullptr);

return;

}

else {

Pop(item->GetSon(), value);

}

}

if (item->GetBrother()) {

if (item->GetBrother()->GetFigure()->GetSide() == value) {

std::shared\_ptr<TTreeItem<T>> ptr = item->GetBrother();

item->SetBrother(item->GetBrother()->GetBrother());

ptr->SetBrother(nullptr);

return;

}

else {

Pop(item->GetBrother(), value);

}

}

}

void TSpace(size\_t n){ // Функция расставляющая пробелы

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

std::cout << " ";

}

template <class T>

void TreeRun(std::ostream &os, std::shared\_ptr<TTreeItem<T>> item, size\_t space){ //Функция обхода дерева для печати

if (item) {

TSpace(space);

//os << \*item << std::endl;

std::cout << "[";

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

std::cout << "]" << "\n";

if (item->GetBrother() != nullptr) {

TreeRun(os, item->GetBrother(),space);

}

if (item->GetSon() != nullptr) {

TreeRun(os, item->GetSon(),space+1);

}

}

}

template <class A>

std::ostream& operator<<(std::ostream &os, const TTree<A> &tree) { //Оператор вывода дерева

std::shared\_ptr<TTreeItem<A>> obj = tree.root;

os << "Printed tree:" << std::endl;

TreeRun(os, obj,1);

return os;

}

template class TTree<Figure>;

template std::ostream &operator<<(std::ostream &os, const TTree<Figure> &obj);

**Tree.h**

#ifndef OOP\_LAB4\_TREE\_H

#define OOP\_LAB4\_TREE\_H

#include "Figure.h"

#include "Item.h"

#include <stdbool.h>

#include <memory>

#include <cstdint>

template <class T> class TTree{

public:

TTree();

int GetNodesOnLevel(int level) {

return GetNodesOnLevel(root, level);

}

std::shared\_ptr<TTreeItem<T>> Push(std::shared\_ptr<TTreeItem<T>> item);

bool empty();

void Pop(size\_t value);

void Pop(std::shared\_ptr<TTreeItem<T>> tree, size\_t value);

template <class A>

friend std::ostream &operator<<(std::ostream &os, const TTree<A> &tree);

virtual ~TTree();

private:

int GetNodesOnLevel(std::shared\_ptr<TTreeItem<T>> item, int level);

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

};

#endif //OOP\_LAB4\_TREE\_H

**Item.h**

#ifndef OOP\_LAB4\_ITEM\_H

#define OOP\_LAB4\_ITEM\_H

#include <stdio.h>

#include <memory>

#include "Figure.h"

template <class T>

class TTreeItem {

public:

TTreeItem(const std::shared\_ptr<T> &figure);

template <class A>

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

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

std::shared\_ptr<TTreeItem<T>> GetBrother();

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

void SetBrother(std::shared\_ptr<TTreeItem<T>> brother);

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

private:

std::shared\_ptr<T> figure;

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

std::shared\_ptr<TTreeItem<T>> brother;

};

#endif //OOP\_LAB4\_ITEM\_H

**Item.cpp**

#include <iostream>

#include <cstdlib>

#include "Item.h"

#include "Figure.h"

template <class T>

TTreeItem<T>::TTreeItem(const std::shared\_ptr<T> &figure) {

this->figure = figure;

this->son = nullptr;

this->brother = nullptr;

std::cout << "Tree item: created" << std::endl;

}

template <class T>

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

return this->figure;

}

template <class T>

std::shared\_ptr<TTreeItem<T>> TTreeItem<T>::GetSon() {

return this->son;

}

template <class T>

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

this->son = son;

}

template <class T>

void TTreeItem<T>::SetBrother(std::shared\_ptr<TTreeItem<T>> brother) {

this->brother = brother;

}

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

{

std::shared\_ptr<TTreeItem<T>> tmp;

if (this->GetSon()) {

return GetSon();

}

else if (this->GetBrother()) {

return GetBrother();

} else if (this->GetParent()) {

tmp = this->GetParent();

while (tmp && !(tmp->GetBrother())) {

tmp = tmp->GetParent();

}

if (!tmp)

return nullptr;

tmp = tmp->GetBrother();

return tmp;

}

return nullptr;

}

template <class T>

std::shared\_ptr<TTreeItem<T>> TTreeItem<T>::GetBrother() {

return this->brother;

}

template <class A>

std::ostream& operator<<(std::ostream &os, const TTreeItem<A> &obj) {

os << "[" << obj.GetFigure() << " " << obj.GetFigure()->Square() << "]";

return os;

}

**Iterator.h**

#ifndef OOP\_LAB4\_ITERATOR\_H

#define OOP\_LAB4\_ITERATOR\_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 const operator++ (int) {

TIterator cur(\*this);

++(\*this);

return cur;

}

bool operator== (const TIterator &it) {

return (cur == it.cur);

}

bool operator!= (const TIterator &it) {

return (cur != it.cur);

}

private:

std::shared\_ptr<N> cur;

};

#endif //OOP\_LAB4\_ITERATOR\_H

**TAllocationBlock.h**

#ifndef OOP\_LAB4\_TALLOCATIONBLOCK\_H

#define OOP\_LAB4\_TALLOCATIONBLOCK\_H

#include <iostream>

#include <cstdlib>

#include "TStack.h"

typedef unsigned char Byte;

class TAllocationBlock

{

public:

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

void \*Allocate();

void Deallocate(void \*ptr);

bool Empty();

size\_t Size();

virtual ~TAllocationBlock();

private:

Byte \*\_used\_blocks;

TStack<void \*>\_free\_blocks;

};

#endif //OOP\_LAB4\_TALLOCATIONBLOCK\_H

**TAllocationBlock.cpp**

#include "TAllocationBlock.h"

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

\_used\_blocks = (Byte \*)malloc(size \* count);

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

void \*ptr =(void \*)malloc(sizeof(void \*));

ptr = \_used\_blocks + i \* size;

\_free\_blocks.Push(ptr);

}

}

void \*TAllocationBlock::Allocate() {

if(!\_free\_blocks.IsEmpty()) {

void \*res = \_free\_blocks.Top();

\_free\_blocks.Pop();

return res;

} else {

throw std::bad\_alloc();

}

}

void TAllocationBlock::Deallocate(void \*ptr) {

\_free\_blocks.Push(ptr);

}

bool TAllocationBlock::Empty() {

return \_free\_blocks.IsEmpty();

}

size\_t TAllocationBlock::Size() {

return \_free\_blocks.GetSize();

}

TAllocationBlock::~TAllocationBlock() {

while(!\_free\_blocks.IsEmpty()) {

\_free\_blocks.Pop();

}

free(\_used\_blocks);

//delete \_free\_blocks;

//delete \_used\_blocks;

}

#ifndef OOP\_LAB4\_TSTACK\_H

#define OOP\_LAB4\_TSTACK\_H

**TStack.h**

#include <iostream>

#include <memory>

#include "TStackItem.h"

template <class T>

class TStack

{

public:

TStack();

virtual ~TStack();

void Push(const T &item);

void Pop();

T &Top();

bool IsEmpty() const;

uint32\_t GetSize() const;

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

private:

TStackItem<T> \*head;

uint32\_t count;

};

#endif //OOP\_LAB4\_TSTACK\_H

**TStack.cpp**

#include <iostream>

#include <memory>

#include "TStackItem.h"

template <class T>

TStackItem<T>::TStackItem(const T &val, TStackItem<T> \*item) {

value = new T(val);

next = item;

}

template <class T>

void TStackItem<T>::Push(const T &val) {

\*value = val;

}

template <class T>

T &TStackItem<T>::Pop() const {

return \*value;

}

template <class T>

void TStackItem<T>::SetNext(TStackItem<T> \*item) {

next = item;

}

template <class T>

TStackItem<T> &TStackItem<T>::GetNext() const {

return \*next;

}

template <class T>

TStackItem<T>::~TStackItem() {

delete value;

}

typedef unsigned char Byte;

template class

TStackItem<void \*>;

Вывод

Аллокатор памяти – часть программы (как прикладной, так и операционной системы), обрабатывающая запросы на выделение и освобождение оперативной памяти или запросы на включение заданной области памяти в адресное пространство процессора. Основное назначение аллокатора памяти в первом смысле – реализация динамической памяти. В языке С динамическое выделение памяти производится через функцию ***malloc***. Программисты должны учитывать последствия динамического выделения памяти и дважды обдумать использование функции ***malloc*** или оператора ***new***. Легко убедить себя, что вы не делаете так уж много аллокаций, а значит большого значения это не имеет, но такой тип мышления распространяется лавиной по всей команде и приводит к медленной смерти. Фрагментация и потери в производительности, связанные с использование динамической памяти, не будучи пресеченными в зародыше, могут иметь катастрофические трудно разрешаемые последствия в вашем дальнейшем цикле разработки. Проекты, где управление и распределение памяти не продумано надлежащим образом, часто страдают от случайных сбоев после длительной сессии из-за нехватки памяти и стоят сотни часов работы программистов, пытающихся освободить память и реорганизовать ее выделение.

Список литературы

1. Бьерн Страуструп — “Язык программирования С++” (Издание 3 специальное).
2. Справочник по языку C++ [Электронный ресурс]. URL: http://cppstudio.com .