**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 6**

Тема: Основы работы с коллекциями: аллокаторы

Студент: Чернобаев Андрей Александрович

Группа: 80-208

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. Постановка задачи

.

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е. равносторонние (кроме трапеции и прямоугольника). Для хранения координат фигур необходимо использовать шаблон std::pair.

Например:

template <class T>

struct Square{

using vertex\_t = std::pair<T,T>;

vertex\_t a,b,c,d;

};

Создать шаблон динамической коллекцию, согласно варианту задания:

1. Коллекция должна быть реализована с помощью умных указателей (std::shared\_ptr, std::weak\_ptr). Опционально использование std::unique\_ptr;

2. В качестве параметра шаблона коллекция должна принимать тип данных;

3. Коллекция должна содержать метод доступа:

o Стек – pop, push, top;

o Очередь – pop, push, top;

o Список, Динамический массив – доступ к элементу по оператору [];

4. Реализовать аллокатор, который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (Динамический массив, Список, Стек, Очередь);

5. Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.

6. Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).

7. Реализовать программу, которая:

o Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;

o Позволяет удалять элемент из коллекции по номеру элемента;

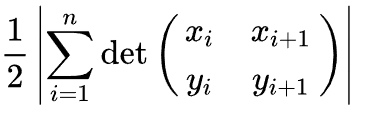
o Выводит на экран введенные фигуры c помощью std::for\_each;

**Варианты заданий (выпуклые равносторонние фигуры вращения):**

|  |  |  |  |
| --- | --- | --- | --- |
| **Вариант** | **Фигура** | **Контейнер** | **Аллокатор** |
| 9. | Треугольник | Список | Динамический массив |

1. Описание программы

Площадь треугольника вычисляется с помощью формулы площади Гаусса:



Программа состоит из 10 файлов:

accessory.h - Заголовочный файл, содержит дополнительные функции программы, облегчающие ввод-вывод.

Состоит из:

* 3-х лямбдо-функций.
* 2-ух функций, реализующих split.
* функция help.

List.h - заголовочный файл для реализации списка согласно заданию

Состоит из:

* класса List
* класса forward\_iterator
* класса Element

Triangle.h - заголовочный файл для треугольника.

Состоит из:

* класса Triangle
* класса перегрузки оператора вывода на печать для класса Triangle

allocator.h - заголовочный файл для аллокатора

Состоит из:

* класса allocator

allocator.cpp - реализация класса allocator

Element.cpp - реализация класса Element из List.h

forward\_iterator.cpp - реализация класса forward\_iterator из List.h

List.cpp - реализация класса List из List.h

Triangle.cpp - реализация класса Triangle из Triangle.h, содержит функцию вычисления детерминанта второго порядка и вычисления площади многоугольника с помощью формулы Гаусса.

1. Набор тестов

Из файла test\_01.txt

0 0 3 3 6 0

0 0 3 3 9 0

0 0 3 3 12 0

delete 1

show

delete 0

show

5 6 3 7 6 5

67 43 23 21 54 78

show

exit

Из файла test\_02.txt

34 6 54 23 13 423

show

4 5 1 12 6 12

show

delete 0

show

0 0 4 4 9 4

0 0 4 4 9 8

5 5 4 12 6 12

delete 2

show

exit

1. Результаты выполнения тестов

|  |  |
| --- | --- |
| input | output |
| 0 0 3 3 6 0  0 0 3 3 9 0  0 0 3 3 12 0  delete 1  show  delete 0  show  5 6 3 7 6 5  67 43 23 21 54 78  show  exit | write exit for exit  write help for help  write delete {index} to remove figure by index  write show to show all figures  input format x\_1 y\_1 x\_2 y\_2 ... x\_n y\_n figure  begin of Triangle  0 0  3 3  6 0  Square is: 9  end of Triangle  begin of Triangle  0 0  3 3  12 0  Square is: 18  end of Triangle  begin of Triangle  0 0  3 3  12 0  Square is: 18  end of Triangle  begin of Triangle  0 0  3 3  12 0  Square is: 18  end of Triangle  begin of Triangle  5 6  3 7  6 5  Square is: 0.5  end of Triangle  begin of Triangle  67 43  23 21  54 78  Square is: 913  end of Triangle |
| 34 6 54 23 13 423  show  4 5 1 12 6 12  show  delete 0  show  0 0 4 4 9 4  0 0 4 4 9 8  5 5 4 12 6 12  delete 2  show  exit | write exit for exit  write help for help  write delete {index} to remove figure by index  write show to show all figures  input format x\_1 y\_1 x\_2 y\_2 ... x\_n y\_n figure  begin of Triangle  34 6  54 23  13 423  Square is: 4348.5  end of Triangle  begin of Triangle  34 6  54 23  13 423  Square is: 4348.5  end of Triangle  begin of Triangle  4 5  1 12  6 12  Square is: 17.5  end of Triangle  begin of Triangle  4 5  1 12  6 12  Square is: 17.5  end of Triangle  begin of Triangle  4 5  1 12  6 12  Square is: 17.5  end of Triangle  begin of Triangle  0 0  4 4  9 4  Square is: 10  end of Triangle  begin of Triangle  5 5  4 12  6 12  Square is: 7  end of Triangle |

1. Листинг программы

из файла **accessory.h**

#ifndef TASK5\_ACCESSORY\_H

#define TASK5\_ACCESSORY\_H

#include <string>

#include <vector>

#include <sstream>

auto string\_to\_int = [](const std::string& s) {return std::stoi(s);};

auto string\_to\_double = [](const std::string& s) {return std::stod(s);};

auto int\_to\_string = [](const int i) {return std::to\_string(i);};

template <typename Out>

void split(const std::string &s, char delim, Out result) {

std::istringstream iss(s);

std::string item;

while (std::getline(iss, item, delim)) {

\*result++ = item;

}

}

std::vector<std::string> split(const std::string &s, char delim) {

std::vector<std::string> elems;

split(s, delim, std::back\_inserter(elems));

return elems;

}

void help(){

std::cout << "write exit for exit" << std::endl;

std::cout << "write help for help" << std::endl;

std::cout << "write delete {index} to remove figure by index" << std::endl;

std::cout << "write show to show all figures" << std::endl;

std::cout << "input format x\_1 y\_1 x\_2 y\_2 ... x\_n y\_n figure" << std::endl;

std::cout << "" << std::endl;

}

#endif //TASK5\_ACCESSORY\_H

Из файла **allocator.cpp**

allocator::allocator(size\_t \_size, size\_t \_count) {

size = \_size;

count = \_count;

free\_blocks.resize(count);

used\_blocks = (char\*) malloc(size \* count);

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

free\_blocks[i] = used\_blocks + i \* size;

}

free\_count = count;

}

void\* allocator::alloc() {

void\* res = nullptr;

if (free\_count > 0){

res = free\_blocks[free\_count - 1];

free\_count--;

}

else {

throw std::overflow\_error("cant't alloc");

}

return res;

}

void allocator::dealloc(void\* ptr) {

free\_blocks[free\_count] = ptr;

free\_count++;

}

allocator::~allocator(){

if (free\_count < count){

std::perror("memory leak");

}

free\_blocks.clear();

free(used\_blocks);

}

Из файла **allocator.h**

#ifndef OOP6\_ALLOCATOR\_H

#define OOP6\_ALLOCATOR\_H

#include <vector>

class allocator {

public:

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

void\* alloc();

void dealloc(void\* ptr);

~allocator();

private:

size\_t size;

size\_t count;

char\* used\_blocks;

std::vector<void\*> free\_blocks;

size\_t free\_count;

};

#include "allocator.cpp"

#endif //OOP6\_ALLOCATOR\_H

Из файла **Element.cpp**

template <class T>

allocator Element<T>::item\_allocator(sizeof(Element<T>),MAX\_ELEMENTS\_COUNT);

template <class T>

forward\_iterator<T> Element<T>::next(){

return forward\_iterator<T>(next\_el.get());

}

template <class T>

void\* Element<T>::operator new (size\_t size) {

return item\_allocator.alloc();

}

template <class T>

void Element<T>::operator delete(void\* p) {

item\_allocator.dealloc(p);

}

Из файла **forward\_iterator.cpp**

template <class T>

forward\_iterator<T>::forward\_iterator(Element<T>\* \_ptr){

ptr = \_ptr;

}

template<class T>

T& forward\_iterator<T>::forward\_iterator::operator\*() {

return ptr->value;

}

template<class T>

forward\_iterator<T>& forward\_iterator<T>::forward\_iterator::operator++() {

if (ptr == nullptr){

throw std::out\_of\_range("out of range");

}

\*this = ptr->next();

return \*this;

}

template<class T>

forward\_iterator<T> forward\_iterator<T>::forward\_iterator::operator++(int) {

forward\_iterator<T> old = \*this;

++\*this;

return old;

}

template<class T>

bool forward\_iterator<T>::forward\_iterator::operator==(const forward\_iterator& Other) const {

return ptr == Other.ptr;

}

template<class T>

bool forward\_iterator<T>::forward\_iterator::operator!=(const forward\_iterator& Other) const {

return ptr != Other.ptr;

}

Из файла **List.cpp**

template<class T>

std::shared\_ptr<Element<T>> pop(std::shared\_ptr<Element<T>> el) {

if (el->next\_el != nullptr) {

el->next\_el = pop(el->next\_el);

return el;

}

return nullptr;

}

template<class T>

std::shared\_ptr<Element<T>> push(std::shared\_ptr<Element<T>> el) {

if (el -> next\_el != nullptr) {

return push(el->next\_el);

}

return el;

}

template <class T>

const size\_t List<T>::length() const{

return size;

}

template <class T>

forward\_iterator<T> List<T>::begin() {

return forward\_iterator<T>(first.get());

}

template <class T>

forward\_iterator<T> List<T>::end() {

return forward\_iterator<T>(nullptr);

}

template <class T>

void List<T>::erase\_by\_it(forward\_iterator<T> it){

if (size == 0){

throw std::invalid\_argument("no elements");

}

if (it.ptr == nullptr){

throw std::invalid\_argument("no element");

}

if (it == begin()){

first = first->next\_el;

if (first != nullptr){

first->prev\_el.reset();

}

size--;

return;

}

if (it == end()){

first = pop(first);

size--;

return;

}

std::shared\_ptr<Element<T>> after\_deleted = it.ptr->next\_el;

std::weak\_ptr<Element<T>> before\_deleted = it.ptr->prev\_el;

after\_deleted->prev\_el = before\_deleted;

before\_deleted.lock()->next\_el = after\_deleted;

size--;

}

template<class T>

void List<T>::erase\_by\_index(const size\_t index) {

forward\_iterator<T> it = begin();

for (size\_t i = 1; i <= index; ++i) {

it++;

}

erase\_by\_it(it);

}

template<class T>

void List<T>::insert\_by\_it(forward\_iterator<T> it,const T& value){

if (first == nullptr){

first = std::shared\_ptr<Element<T>>(new Element<T>{value});

}

else {

auto temp = std::shared\_ptr<Element<T>>(new Element<T>{value});

if (it == begin()){

temp->next\_el = first;

first->prev\_el = temp;

first = temp;

}

else if (it.ptr == nullptr){

temp->prev\_el = push(first);

push(first)->next\_el = temp;

}

else {

temp->prev\_el = it.ptr->prev\_el;

temp->next\_el = it.ptr->prev\_el.lock()->next\_el;

it.ptr->prev\_el = temp;

temp->prev\_el.lock()->next\_el = temp;

}

}

size++;

}

template <class T>

void List<T>::insert\_by\_index(const size\_t n,const T& value){

forward\_iterator<T> it = begin();

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

it++;

}

insert\_by\_it(it, value);

}

template<class T>

List<T>& List<T>::operator=(List<T>&& Other){

size = Other.size;

first = std::move(Other.first);

}

template<class T>

T& List<T>::operator[](const size\_t index) const {

if (index < 0 || index >= size) {

throw std::out\_of\_range("out of range");

}

forward\_iterator<T> it = begin();

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

it++;

}

return \*it;

}

Из файла **List.h**

#ifndef OOP5\_LIST\_H

#define OOP5\_LIST\_H

#include <memory>

#include "allocator.h"

#define MAX\_ELEMENTS\_COUNT 1000

template <class T>

class forward\_iterator;

template <class T>

class Element;

template <class T>

class List {

public:

List() = default;

const size\_t length() const;

forward\_iterator<T> begin();

forward\_iterator<T> end();

void erase\_by\_it(forward\_iterator<T> it);

void erase\_by\_index(const size\_t n);

void insert\_by\_it(forward\_iterator<T> it,const T& value);

void insert\_by\_index(const size\_t n,const T& value);

T& operator[](const size\_t index) const;

List<T>& operator=(List<T>&& Other);

private:

size\_t size = 0;

std::shared\_ptr<Element<T>> first = nullptr;

};

template <class T>

class forward\_iterator {

public:

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = std::ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

explicit forward\_iterator(Element<T>\* ptr);

T& operator\*();

forward\_iterator& operator++();

forward\_iterator operator++(int);

bool operator== (const forward\_iterator& other) const;

bool operator!= (const forward\_iterator& other) const;

private:

Element<T>\* ptr;

friend List<T>;

};

template <class T>

class Element {

public:

T value;

std::shared\_ptr<Element> next\_el;

std::weak\_ptr<Element> prev\_el ;

forward\_iterator<T> next();

static allocator item\_allocator;

void\* operator new(size\_t size);

void operator delete(void\* p);

};

#include "List.cpp"

#include "Element.cpp"

#include "forward\_iterator.cpp"

#endif //OOP5\_LIST\_H

Из файла **main.cpp**

#include <iostream>

#include <vector>

#include <algorithm>

#include "List.h"

#include "Triangle.h"

#include "accessory.h"

using triangle\_type = int;

auto main() -> int {

std::string cmd;

std::vector<std::string> out;

List<Triangle<triangle\_type>> list;

help();

while (true){

getline(std::cin, cmd);

std::vector<std::string> cmd\_array = split(cmd,' ');

if (cmd\_array[0] == "exit"){

break;

}

else if (cmd\_array[0] == "help"){

help();

}

else if (cmd\_array[0] == "delete"){

size\_t index = string\_to\_int(cmd\_array[1]);

list.erase\_by\_index(index);

}

else if (cmd\_array[0] == "show"){

std::for\_each(list.begin(), list.end(), [](Triangle<triangle\_type>& T) { std::cout << T; });

}

else {

std::vector<std::string> array = split(cmd,' ');

std::vector<std::pair<triangle\_type ,triangle\_type>> numbers\_double(EDGES\_COUNT);

int j = 0;

for(size\_t i = 0;i < numbers\_double.size() \* 2 ; i += 2){

numbers\_double[j] = {string\_to\_int(array[i]),string\_to\_int(array[i+1])};

j++;

}

Triangle<triangle\_type> tr(numbers\_double);

list.insert\_by\_index(list.length(),tr);

}

}

return 0;

}

Из файла **Triangle.cpp**

double get\_det(const double x1, const double y1,

const double x2, const double y2){

return x1 \* y2 - x2 \* y1;

}

template <class T>

double gauss\_area(const std::vector<std::pair<T,T>>& nodes, size\_t size){

double out = 0.0;

auto count = size;

for(size\_t i = 0; i < (count - 1); i++){

out += get\_det(static\_cast<double>(nodes[i].first),static\_cast<double>(nodes[i].second),

static\_cast<double>(nodes[i+1].first),static\_cast<double>(nodes[i+1].second));

}

out += get\_det(static\_cast<double>(nodes[count-1].first),static\_cast<double>(nodes[count-1].second),

static\_cast<double>(nodes[0].first),static\_cast<double>(nodes[0].second));

return fabs(0.5 \* out);

}

template<class T>

double Triangle<T>::get\_square() const {

return gauss\_area<T>(edges,EDGES\_COUNT);

}

template<class T>

Triangle<T>::Triangle(vertex\_t<T>& \_a, vertex\_t<T>& \_b, vertex\_t<T>& \_c) {

a = \_a;

b = \_b;

c = \_c;

edges.resize(EDGES\_COUNT);

edges[0] = a;

edges[1] = b;

edges[2] = c;

if (get\_square() == 0){

throw std::logic\_error("It's not triangle");

}

}

template<class T>

Triangle<T>::Triangle(std::vector<vertex\_t<T>> \_vec) {

edges = \_vec;

if (get\_square() == 0){

throw std::logic\_error("It's not triangle");

}

}

template <class T>

std::ostream& operator << (std::ostream& out, const vertex\_t<T>& p){

out << p.first << " " << p.second << std::endl;

return out;

}

template <class T>

std::ostream &operator<<(std::ostream& out, const Triangle<T>& tr) {

out << "begin of Triangle" << std::endl;

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

out << tr.edges[i];

}

out << "Square is: " << tr.get\_square() << std::endl;

out << "end of Triangle" << std::endl;

return out;

}

Из файла **Triangle.h**

#ifndef OOP6\_TRIANGLE\_H

#define OOP6\_TRIANGLE\_H

#include <vector>

#include <cmath>

#define EDGES\_COUNT 3

template <class T>

using vertex\_t = std::pair<T,T>;

template <class T>

class Triangle {

public:

vertex\_t<T> a,b,c;

std::vector<vertex\_t<T>> edges;

double get\_square() const;

explicit Triangle(vertex\_t<T>& \_a,vertex\_t<T>& \_b,vertex\_t<T>& \_c);

explicit Triangle(std::vector<vertex\_t<T>> \_vec);

};

template <class T>

std::ostream& operator << (std::ostream& out, const Triangle<T>& tr);

#include "Triangle.cpp"

#endif //OOP6\_TRIANGLE\_H

1. Выводы.

Я научился реализовывать собственный аллокатор и использовать его в коллекциях.

1. Список использованных источников.

1 std::shared\_ptr [Электронный ресурс].

URL: https://en.cppreference.com/w/cpp/memory/shared\_ptr

2 Templates [Электронный ресурс].

URL: https://www.cplusplus.com/doc/oldtutorial/templates/

3 std::allocator [Электронный ресурс].

URL: http://www.cplusplus.com/reference/memory/allocator/

4 Allocator (C++) [Электронный ресурс].

URL: https://en.wikipedia.org/wiki/Allocator\_(C%2B%2B)

5 operator new, operator new[] [Электронный ресурс].

URL: https://en.cppreference.com/w/cpp/memory/new/operator\_new