**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: Кочисов Алексей

Группа: М8О-201Б-20

Преподаватель: Чернышев Л. Н.

Дата:

Оценка:

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. равносторонними (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:
   * Стек – pop, push, top;
   * Очередь – pop, push, top;
   * 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:
   * Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;
   * Позволяет удалять элемент из коллекции по номеру элемента;
   * Выводит на экран введенные фигуры c помощью std::for\_each;

*Вариант 3: Прямоугольник — стек — стек*

1. **Описание программы**

Задание разбито на несколько файлов: *main.cpp, Stack.h, Rectangle.h Allocator.h.*

В *main.cpp* находится набор тестов, нужных для тестирования программы.

В *Rectangle.h* находится класс rectangle, в котором есть координаты фигуры и заданы стороны. Также есть метод, вычисляющий площадь фигуры

В *Stack.h* написана коллекция, с которой я работаю. Для начала делается структура *Node,* в которой содержатся данные элемента структуры и ссылка на предыдущий элемент стека, написанный при помощи умных указателей. Далее перегружается операторы для того, чтобы была совместимость со стандартными алгоритмами. Далее идет описание самого стека. В нем данные хранятся при помощи указателя shared\_ptr<Node>. Это сделано для того, чтобы можно было хранить в стеке данные мне фигуры. Далее идет описание стандартных методов стека. Чего-то необычного там нет.

Напоследок идет класс итератора, который написан внутри Stack. Это было сделано для удобства обращения к итератору. Далее идут using, задающие характеристики нашего итератора. Это надо, опять же, для совместимости со стандартными алгоритмами. В нем же идет перегрузка операторов, нужных для совместимости

В файле Allocator.h написан класс аллокатора, реализованный на стеке. Есть несколько особенностей в данном классе. Заключаются эти особенности в том, что *constructor* сделан так, что в *OTHER\_T* содержится тип аргументов, а в …*ARGS* сами аргументы. Это сделано потому, что аллокатор не может самостоятельно вызывать конструктор. Вместо него вызывается данный метод.

*deallocate* – пустой метод. Задается вопрос: зачем нужна эта функция? Она необходима, чтобы была совместимость со стандартными алгоритмами, но в том же время у аллокатора нет деструктора. Поэтому этот метод – пустой.

1. **Наборы и результаты выполнения тестов**

*Тест 1:*

Allocatind 1 bytes in block with 256 free bytes

Allocatind 2 bytes in block with 255 free bytes

Allocatind 4 bytes in block with 253 free bytes

Allocatind 8 bytes in block with 249 free bytes

Allocatind 16 bytes in block with 241 free bytes

0

1

2

3

4

5

6

7

8

9

Allocatind 1 bytes in block with 256 free bytes

Allocatind 1 bytes in block with 255 free bytes

Allocatind 1 bytes in block with 254 free bytes

Allocatind 1 bytes in block with 253 free bytes

Allocatind 1 bytes in block with 252 free bytes

Allocatind 1 bytes in block with 251 free bytes

Allocatind 1 bytes in block with 250 free bytes

Allocatind 1 bytes in block with 249 free bytes

Allocatind 1 bytes in block with 248 free bytes

Allocatind 1 bytes in block with 247 free bytes

0

1

2

3

4

5

6

7

8

9

Allocatind 1 bytes in block with 256 free bytes

Allocatind 1 bytes in block with 255 free bytes

Allocatind 1 bytes in block with 254 free bytes

Allocatind 1 bytes in block with 253 free bytes

Allocatind 1 bytes in block with 252 free bytes

Allocatind 1 bytes in block with 251 free bytes

Allocatind 1 bytes in block with 250 free bytes

Allocatind 1 bytes in block with 249 free bytes

Allocatind 1 bytes in block with 248 free bytes

Allocatind 1 bytes in block with 247 free bytes

0

1

2

3

4

5

6

7

8

9

Input number of rectangle in stack

1

Enter index to insert a rectangle

2

Enter your point and sides

First is point. Enter x and y.

Second are sides. Enter a and b

1 1

2 3

terminate called after throwing an instance of 'std::runtime\_error'

what(): Iterator points to nullptr!

Process finished with exit code 3

*Тест 2:*

Allocatind 1 bytes in block with 256 free bytes

Allocatind 2 bytes in block with 255 free bytes

Allocatind 4 bytes in block with 253 free bytes

Allocatind 8 bytes in block with 249 free bytes

Allocatind 16 bytes in block with 241 free bytes

0

1

2

3

4

5

6

7

8

9

Allocatind 1 bytes in block with 256 free bytes

Allocatind 1 bytes in block with 255 free bytes

Allocatind 1 bytes in block with 254 free bytes

Allocatind 1 bytes in block with 253 free bytes

Allocatind 1 bytes in block with 252 free bytes

Allocatind 1 bytes in block with 251 free bytes

Allocatind 1 bytes in block with 250 free bytes

Allocatind 1 bytes in block with 249 free bytes

Allocatind 1 bytes in block with 248 free bytes

Allocatind 1 bytes in block with 247 free bytes

0

1

2

3

4

5

6

7

8

9

Allocatind 1 bytes in block with 256 free bytes

Allocatind 1 bytes in block with 255 free bytes

Allocatind 1 bytes in block with 254 free bytes

Allocatind 1 bytes in block with 253 free bytes

Allocatind 1 bytes in block with 252 free bytes

Allocatind 1 bytes in block with 251 free bytes

Allocatind 1 bytes in block with 250 free bytes

Allocatind 1 bytes in block with 249 free bytes

Allocatind 1 bytes in block with 248 free bytes

Allocatind 1 bytes in block with 247 free bytes

0

1

2

3

4

5

6

7

8

9

Input number of rectangle in stack

2

Enter index to insert a rectangle

0

Enter your point and sides

First is point. Enter x and y.

Second are sides. Enter a and b

1 1

2 3

Allocatind 24 bytes in block with 256 free bytes

Enter index to insert a rectangle

1

Enter your point and sides

First is point. Enter x and y.

Second are sides. Enter a and b

2 3

10 10

Allocatind 24 bytes in block with 232 free bytes

Your entering:

Coordinates: {(1, 1); (1, 3); (4, 3); (4, 1)}

Coordinates: {(2, 3); (2, 13); (12, 13); (12, 3)}

Enter index to erase from stack

0

After erase:

Coordinates: {(2, 3); (2, 13); (12, 13); (12, 3)}

Process finished with exit code 0

1. **Листинг программы**

**main.cpp**

#include "Stack.h"  
#include <algorithm>  
#include <vector>  
#include <set>  
#include <list>  
#include <iostream>  
#include <exception>  
  
  
const std::size\_t BLOCK\_SIZE = 256;  
  
int main() {  
 std::vector<int, linear\_allocator\_t<int, BLOCK\_SIZE> > vec;  
 for (int i = 0; i < 10; ++i) {  
 vec.push\_back(i);  
 }  
 for (int i = 0; i < 10; ++i) {  
 vec.pop\_back();  
 }  
 for (int i = 0; i < 10; ++i) {  
 vec.push\_back(i);  
 }  
 for (const int & elem : vec) {  
 std::cout << elem << std::endl;  
 }  
  
 std::set<int, std::less<int>, linear\_allocator\_t<int, BLOCK\_SIZE> > s;  
 for (int i = 0; i < 10; ++i) {  
 s.insert(i);  
 }  
 for (int elem : s) {  
 std::cout << elem << std::endl;  
 }  
  
 std::list<int, linear\_allocator\_t<int, BLOCK\_SIZE> > l;  
 for (int i = 0; i < 10; ++i) {  
 l.push\_back(i);  
 }  
 for (int elem : l) {  
 std::cout << elem << std::endl;  
 }  
  
 /\* I \*/  
  
 auto Print = [](const auto & elem) {  
 std::cout << elem << std::endl;  
 };  
 size\_t n;  
 std::cout << "Input number of rectangle in stack" << std::endl;  
 std::cin >> n;  
 int cordX, cordY, sideA, sideB;  
 Stack< Rectangle<int>, linear\_allocator\_t< Rectangle<int>, BLOCK\_SIZE > > st;  
 for (size\_t i = 0; i < n; ++i) {  
 size\_t n;  
 std::cout << "Enter index to insert a rectangle" << std::endl;  
 std::cin >> n;  
 std::cout << "Enter your point and sides\n"  
 << "First is point. Enter x and y.\n"  
 << "Second are sides. Enter a and b" << std::endl;  
 std::cin >> cordX >> cordY >> sideA >> sideB;  
 try {  
 auto it = st.begin();  
 while (n--) {  
 ++it;  
 }  
 st.insert(it, Rectangle<int>(std::pair<int, int>(cordX, cordY), sideA, sideB));  
 } catch (std::runtime\_error & exception) {  
 std::terminate();  
 }  
 }  
 std::cout << "Your entering:" << std::endl;  
 std::for\_each(st.begin(), st.end(), Print);  
 std::cout << "Enter index to erase from stack" << std::endl;  
 std::cin >> n;  
 try {  
 auto it = st.begin();  
 while (n > 1) {  
 ++it;  
 --n;  
 }  
 st.erase(it);  
 } catch (std::runtime\_error & exception) {  
 std::terminate();  
 }  
 std::cout << "After erase:" << std::endl;  
 std::for\_each(st.begin(), st.end(), Print);  
 return 0;  
}

**Stack.h**

#ifndef OOP\_EXERCISE\_05\_STACK\_H  
#define OOP\_EXERCISE\_05\_STACK\_H  
  
#include "Rectangle.h"  
#include "Allocator.h"  
  
template<class T, class ALLOCATOR>  
class Stack {  
private:  
 struct TNode;  
  
 using allocator\_type = typename ALLOCATOR::template rebind<TNode>::other;  
  
 struct deleter {  
 allocator\_type stack\_node\_deleter;  
  
 deleter() : stack\_node\_deleter() {};  
 deleter(allocator\_type\* another\_deleter) : stack\_node\_deleter(another\_deleter) {}  
  
 /\* std::shared\_ptr uses operator() to delete memory \*/  
 void operator() (void\* ptr) {  
 stack\_node\_deleter.deallocate((TNode\*)ptr, 1);  
 }  
 };  
  
 struct TNode {  
 T data;  
 std::shared\_ptr<TNode> next;  
  
 TNode() noexcept : data(), next(nullptr) {};  
 explicit TNode(const T & elem) noexcept : data(elem), next(nullptr) {}  
  
 friend bool operator != (const TNode & lhs, const TNode & rhs) {  
 return &lhs.data != &rhs.data;  
 }  
  
 friend bool operator == (const TNode & lhs, const TNode & rhs) {  
 return &lhs.data == &rhs.data;  
 }  
  
 friend std::ostream & operator << (std::ostream & out, const TNode & node) {  
 out << node.data;  
 return out;  
 }  
 };  
  
public:  
 class iterator {  
 private:  
 std::shared\_ptr<TNode> ptr;  
 public:  
 using iterator\_category = std::forward\_iterator\_tag;  
 using difference\_type = std::ptrdiff\_t;  
 using value\_type = T;  
 using pointer = T\*;  
 using reference = T&;  
 using const\_reference = const T&;  
  
 iterator() : ptr(nullptr) {}  
 iterator(const std::shared\_ptr<TNode> & another\_ptr) : ptr(another\_ptr) {}  
  
 bool is\_null() {  
 return ptr == nullptr;  
 }  
  
 void unvalidate() {  
 ptr = nullptr;  
 }  
  
 iterator & operator ++ () {  
 if (this->ptr != nullptr) {  
 this->ptr = this->ptr->next;  
 return \*this;  
 } else {  
 throw(std::runtime\_error("Iterator points to nullptr!"));  
 }  
 }  
  
 bool operator != (const iterator & other\_iterator) {  
 return &other\_iterator.ptr->data != &this->ptr->data;  
 }  
  
 friend std::ostream & operator << (std::ostream & out, const iterator & it) {  
 out << \*(it.ptr);  
 return out;  
 }  
  
 TNode & operator \* () {  
 return \*ptr;  
 }  
 };  
  
private:  
 std::shared\_ptr<TNode> top\_node;  
 deleter stack\_deleter;  
  
public:  
 Stack() noexcept : top\_node() {};  
  
 iterator begin() {  
 return iterator(top\_node);  
 }  
  
 iterator end() {  
 return iterator(nullptr);  
 }  
  
 void pop() {  
 if (top\_node) {  
 top\_node = top\_node->next;  
 } else {  
 throw(std::runtime\_error("Stack is empty!"));  
 }  
 }  
  
 void push(const T & elem) {  
 TNode\* new\_node = stack\_deleter.stack\_node\_deleter.allocate(sizeof(TNode));  
 stack\_deleter.stack\_node\_deleter.construct(new\_node, elem);  
 std::shared\_ptr<TNode> new\_node\_shared(new\_node, stack\_deleter);  
 new\_node\_shared->next = top\_node;  
 top\_node = new\_node\_shared;  
 }  
  
 T top() {  
 if (top\_node) {  
 return top\_node->data;  
 } else {  
 throw(std::runtime\_error("Stack is empty!"));  
 }  
 }  
  
 void erase(iterator it) {  
 if (it.is\_null()) {  
 throw(std::runtime\_error("Iterator points to nullptr!"));  
 } else {  
 if (\*it == \*top\_node) {  
 top\_node = top\_node->next;  
 } else {  
 std::shared\_ptr<TNode> prev\_node = top\_node;  
 while (\*prev\_node->next != \*it) {  
 prev\_node = prev\_node->next;  
 }  
 prev\_node->next = prev\_node->next->next;  
 (\*it).next = nullptr;  
 }  
 it.unvalidate();  
 }  
 }  
  
 void insert(iterator it, const T & elem) {  
 TNode\* new\_node = stack\_deleter.stack\_node\_deleter.allocate(sizeof(TNode));  
 stack\_deleter.stack\_node\_deleter.construct(new\_node, elem);  
 std::shared\_ptr<TNode> new\_node\_shared(new\_node, stack\_deleter);  
 if (top\_node) {  
 if (\*it == \*top\_node) {  
 new\_node\_shared->next = top\_node;  
 top\_node = new\_node\_shared;  
 it.unvalidate();  
 return;  
 }  
 std::shared\_ptr<TNode> prev\_node = top\_node;  
 while (\*prev\_node->next != \*it) {  
 prev\_node = prev\_node->next;  
 }  
 if (it.is\_null()) {  
 prev\_node->next = new\_node\_shared;  
 } else {  
 new\_node\_shared->next = prev\_node->next;  
 prev\_node->next = new\_node\_shared;  
 // std::swap(prev\_node->data, prev\_node->next->data);  
 }  
 } else {  
 top\_node = new\_node\_shared;  
 }  
 it.unvalidate();  
 }  
};  
  
#endif //OOP\_EXERCISE\_05\_STACK\_H

**Rectangle.h**

#ifndef OOP\_EXERCISE\_05\_RECTANGLE\_H  
#define OOP\_EXERCISE\_05\_RECTANGLE\_H  
  
#include <iostream>  
#include <memory>  
#include <exception>  
  
template<class T>  
struct Rectangle{  
 using vertex\_t = std::pair<T,T>;  
 vertex\_t point;  
 T a;  
 T b;  
  
 Rectangle(const vertex\_t& \_point, const T& \_a, const T& \_b ):a(\_a), point(\_point),b(\_b){}  
  
 T area() const {  
 return a\*b;  
 }  
};  
  
template<class T>  
std::ostream& operator << (std::ostream& cout, const std::pair<T,T>& val){  
 cout << "(" << val.first << ", " << val.second << ")";  
 return cout;  
}  
  
template<class T>  
std::ostream& operator << (std::ostream& cout, const Rectangle<T>& obj){  
 cout << "Coordinates: {"  
 << std::pair<T,T>(obj.point.first,obj.point.second)<< "; "  
 << std::pair<T,T>(obj.point.first,obj.point.second + obj.a)<< "; "  
 << std::pair<T,T>(obj.point.first + obj.b,obj.point.second + obj.a)<< "; "  
 << std::pair<T,T>(obj.point.first + obj.b,obj.point.second)<< "}\n ";  
  
 return cout;  
}  
  
#endif //OOP\_EXERCISE\_05\_RECTANGLE\_H

**Allocator.h**

#ifndef ALLOCATOR\_HPP  
#define ALLOCATOR\_HPP  
#include <stack>  
  
template<class T, std::size\_t BLOCK\_SIZE>  
class linear\_allocator\_t {  
private:  
 std::stack<T\*> st;  
 T\* buffer;  
public:  
 using allocator\_type = linear\_allocator\_t;  
 using value\_type = T;  
 using pointer = T\*;  
 using reference = T&;  
 using const\_reference = const T&;  
 using size\_type = std::size\_t;  
  
 T\* allocate(const std::size\_t & n) {  
 if (buffer == nullptr) {  
 buffer = new T[BLOCK\_SIZE];  
 for (std::size\_t i = 0; i < BLOCK\_SIZE; ++i) {  
 st.push(&buffer[i]);  
 }  
 }  
 printf("Allocatind %lu bytes in block with %lu free bytes\n", n, st.size());  
 if (st.size() < n) {  
 throw(std::bad\_alloc());  
 } else {  
 T\* p = st.top();  
 for (std::size\_t i = 0; i < n; ++i) {  
 st.pop();  
 }  
 return p;  
 }  
 }  
  
 template<class OTHER\_T, class... ARGS>  
 void construct(OTHER\_T\* p, ARGS... arguments) {  
 \*p = OTHER\_T(std::forward<ARGS>(arguments)...);  
 }  
  
 void deallocate(pointer, std::size\_t) {  
 ;  
 }  
  
 linear\_allocator\_t() : st(), buffer(nullptr) {  
 static\_assert(BLOCK\_SIZE > 0);  
 }  
  
 explicit linear\_allocator\_t(const linear\_allocator\_t<T, BLOCK\_SIZE> & another\_allocator) : linear\_allocator\_t() {  
 buffer = new T[BLOCK\_SIZE];  
 for (std::size\_t i = 0; i < BLOCK\_SIZE; ++i) {  
 buffer[i] = another\_allocator.buffer[i];  
 st.push(&buffer[i]);  
 }  
 }  
  
 ~linear\_allocator\_t() {  
 delete[] buffer;  
 }  
  
 template<class OTHER\_T>  
 class rebind {  
 public:  
 using other = linear\_allocator\_t<OTHER\_T, BLOCK\_SIZE>;  
 };  
};  
  
#endif /\* ALLOCATOR\_HPP \*/

1. **Вывод**

В ходе выполнения работы я узнал, как реализовать аллокатор на языке С++, чтобы он был совместим со стандартными функциями. Реализовал свой линейный аллокатор и стек, работающий с ним.