**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: Ляшун Дмитрий Сергеевич

Группа: 80-207

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. **Постановка задачи**

Разработать программу на языке C++ согласно варианту задания. Программа на C++ должна собираться с помощью системы сборки CMake. Программа должна получать данные из стандартного ввода и выводить данные в стандартный вывод.

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 №24:**

|  |  |  |
| --- | --- | --- |
| **Фигура** | **Контейнер** | **Аллокатор** |
| 8-угольник | Очередь | Очередь |

1. **Описание программы**

Для реализации фигура 8-угольник опишем шаблонную структуру **Octagon** со следующими полями:

|  |  |  |
| --- | --- | --- |
| **Имя** | **Тип** | **Назначение** |
| vec | vector\_t (pair<T, T>) | Координаты вектора от центра к середине одной из сторон восьмиугольника. |
| vx | vertex\_t (pair<T, T>) | Координаты центра восьмиугольника. |
| square | double | Значение площади восьмиугольника. |

Для работы с восьмиугольником **Octagon** будут определены следующие функции:

|  |  |
| --- | --- |
| **Функция** | **Назначение** |
| Octagon(const vertex\_t& vx, const vector\_t& vec) | Конструктор, создающий восьмиугольник по координатам центра vx и вектора vec радиуса описанной окружности, также при создании вычисляет площадь полученной фигуры. |
| void print() const | Процедура вывода восьмиугольника. |
| template<class T>  double Length(const pair<T, T>& vec) | Шаблонная функция, вычисляющая длину заданного вектора vec. |

Коллекция очередь будет реализована посредством двусвязного списка. Для этого сперва опишем вспомогательную шаблонную структуру **Node** со следующими полями:

|  |  |  |
| --- | --- | --- |
| **Имя** | **Тип** | **Назначение** |
| next | shared\_ptr<Node> | Указатель на следующий узел в списке. |
| prev | shared\_ptr<Node> | Указатель на предыдущий узел в списке. |
| elem | T (template<class T>) | Хранимый элемент узла. |

Затем определим шаблонный класс очереди **Queue** со следующими полями:

|  |  |  |
| --- | --- | --- |
| **Имя** | **Тип** | **Назначение** |
| begin\_ptr | shared\_ptr<Node<T> > | Указатель на начало очереди. |
| end\_ptr | shared\_ptr<Node<T> > | Указатель на конец очереди. |
| size | size\_t | Размер очереди. |

Для работы со очередью **Queue** также опишем необходимые методы:

|  |  |
| --- | --- |
| **Функция** | **Назначение** |
| size\_t Size() | Возвращает значение текущего размера очереди. |
| auto Begin() | Возвращает **forward\_iterator** на начало очереди. |
| auto End() | Возвращает **forward\_iterator** на конец очереди. |
| void Insert(auto iterator, const T& elem) | Процедура вставки элемента elem в очередь в позицию, на которую указывает итератор iterator. |
| void Erase(auto iterator) | Процедура удаления элемента из очереди в позиции, на которую указывает итератор iterator |
| T Front() | Возвращает элемент, находящийся в начале очереди. |
| void Push(const T) | Процедура вставки элемента в конец очереди. |
| void Pop() | Процедура извлечения элемента из начала очереди. |
| static allocator\_type get\_allocator() | Возвращает аллокатор очереди, который используется для выделения/освобождения памяти при добавлении/удалении элементов из очереди. |

Также для более удобной работы с очередью был реализован **forward\_iterator** со следующими полями:

|  |  |  |
| --- | --- | --- |
| **Имя** | **Тип** | **Назначение** |
| ptr | pointer (weak\_ptr<Node<T> >) | Указатель на узел в списке очереди. |
| index | size\_t | Индекс позиции, на которую указывает итератор. |
| size | size\_t | Размер рассматриваемой очереди. |

Для выделения памяти под элементы очереди был описан шаблонный класс аллокатора **my\_allocator** со следующими полями:

|  |  |  |
| --- | --- | --- |
| **Имя** | **Тип** | **Назначение** |
| buffer | T\* | Указатель на область памяти, где хранятся элементы очереди. |
| q | std::queue<T\*> | Очередь с указателями на свободные области памяти в buffer. |

В свою очередь для работы с аллокатором **my\_allocator** были определены следующие методы:

|  |  |
| --- | --- |
| **Функция** | **Назначение** |
| T \*allocate(size\_t n) | Возвращает указатель на область памяти под n элементов типа T. |
| void deallocate(T\* pointer, size\_t n) | Освобождает память из массива pointer размера n с элементами типа T. |
| template <typename U, typename... Args>  void construct(U \*p, Args &&... args) | Вызывает конструктор при выделении памяти под массив p с элементами типа U и передает в него аргументы args. |
| void destroy(T\* p) | Вызывает деструктор при освобождении массива p с элементами типа T. |

1. **Руководство по использованию программы**

**oop\_exercise\_06** – исполняемый файл программы.

Взаимодействие пользователя с программой было реализовано посредством интерактивного меню со следующими поддерживаемыми командами:

1. Добавление восьмиугольника в коллекцию, для этого указывается координаты его центра и вектора радиуса описанной окружности, а также позиция вставки.
2. Удаление восьмиугольника из коллекции, необходимо указать позицию удаления.
3. Вывод всех восьмиугольников в коллекции.
4. Выход из программы.

**Входные данные**

Номера вызываемых команд, позиция проведения операции (для вставки или удаления), координаты центра восьмиугольника и вектора радиуса описанной окружности (для вставки).

**Выходные данные**

В ответ на каждую команду выводится её результат выполнения.

**Тестовые примеры**

> 1

Введите восьмиугольник (коорд. центра и вектор радиуса описан. окружн.): 1 2 3 4

Введите позицию добавления: 0

Ошибка! Произошел выход за границы очереди!

> 1

Введите восьмиугольник (коорд. центра и вектор радиуса описан. окружн.): 1 2 3 4

Введите позицию добавления: 1

Восьмиугольник добавлен в коллекцию.

> 1

Введите восьмиугольник (коорд. центра и вектор радиуса описан. окружн.): 3 3 3 3

Введите позицию добавления: 1

Восьмиугольник добавлен в коллекцию.

> 1

Введите восьмиугольник (коорд. центра и вектор радиуса описан. окружн.): 9 2 1 3

Введите позицию добавления: 1

Восьмиугольник добавлен в коллекцию.

> 3

Восьмиугольник №1 : центр: (9, 2), вектор R описан. окружн.: (1, 3), S = 28.2843

Восьмиугольник №2 : центр: (3, 3), вектор R описан. окружн.: (3, 3), S = 50.9117

Восьмиугольник №3 : центр: (1, 2), вектор R описан. окружн.: (3, 4), S = 70.7107

> 1

Введите восьмиугольник (коорд. центра и вектор радиуса описан. окружн.): 10 11 4 3

Введите позицию добавления: 2

Восьмиугольник добавлен в коллекцию.

> 3

Восьмиугольник №1 : центр: (9, 2), вектор R описан. окружн.: (1, 3), S = 28.2843

Восьмиугольник №2 : центр: (10, 11), вектор R описан. окружн.: (4, 3), S = 70.7107

Восьмиугольник №3 : центр: (3, 3), вектор R описан. окружн.: (3, 3), S = 50.9117

Восьмиугольник №4 : центр: (1, 2), вектор R описан. окружн.: (3, 4), S = 70.7107

> 2

Введите позицию удаления: 0

Ошибка! Произошел выход за границы очереди!

> 2

Введите позицию удаления: 5

Ошибка! Выход за границы очереди!

> 2

Введите позицию удаления: 2

Восьмиугольник удален из коллекции.

> 2

Введите позицию удаления: 3

Восьмиугольник удален из коллекции.

> 3

Восьмиугольник №1 : центр: (9, 2), вектор R описан. окружн.: (1, 3), S = 28.2843

Восьмиугольник №2 : центр: (3, 3), вектор R описан. окружн.: (3, 3), S = 50.9117

> 4

**4. Листинг программы**

Исходный код программы состоит из трех файлов: **octagon.hpp** – содержит реализацию структуры **Octagon**; **queue.hpp** – содержит реализацию структуры **Node**, класса очереди **Queue**, класса итератора **forward\_iterator** и реализацию класса аллокатора **my\_allocator** для перемещения по очереди и доступа к содержимому; **main.cpp** – основная программа. **CMakeLists.txt** – содержит код для утилиты cmake, которая проводит сборку программы из исходного кода.

**Содержимое файла octagon.hpp**

/\* Ляшун Дмитрий Сергеевич, группа М8О-207Б-19 \*/

#include <cmath>

#include <iostream>

using namespace std;

template<class T>

double Length(const pair<T, T>& vec) {

return sqrt(vec.first\*vec.first+vec.second\*vec.second);

}

template<class T>

struct Octagon {

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

using vector\_t = pair<T, T>;

Octagon() = default;

~Octagon() = default;

Octagon(const vertex\_t& vx, const vector\_t& vec) : vec(vec), vx(vx) {

square = 4.0\*sin(acos(-1)/4.0)\*Length(vec)\*Length(vec);

}

void print() const {

cout << "центр: (" << vx.first << ", " << vx.second << "), ";

cout << "вектор R описан. окружн.: (" << vec.first << ", " << vec.second << "), ";

cout << "S = " << square << endl;

}

vertex\_t vec;

vector\_t vx;

double square;

};

**Содержимое файла queue.hpp**

/\* Ляшун Дмитрий Сергеевич, группа М8О-207Б-19 \*/

#include <queue>

#include <cassert>

#include <memory>

#include <functional>

using namespace std;

template<class T, size\_t SIZE>

class my\_allocator {

private:

T\* buffer;

std::queue<T\*> q;

public:

using value\_type = T;

using pointer = T \*;

using const\_pointer = const T \*;

using size\_type = std::size\_t;

template <class U>

struct rebind {

using other = my\_allocator<U, SIZE>;

};

my\_allocator() : buffer(nullptr) {

static\_assert(SIZE > 0);

}

~my\_allocator() {

delete[] buffer;

}

T \*allocate(size\_t n) {

if (buffer == nullptr) {

buffer = new T[SIZE];

for (int i = 0; i < SIZE; ++i) {

q.push(&buffer[i]); // buffer + i

}

}

else if (q.size() < n) {

return throw(std::bad\_alloc());

}

else {

T\* res = q.front();

for (int i = 0; i < n; ++i) {

q.pop();

}

return res;

}

}

void deallocate(T\* pointer, size\_t n) {

for (int i = 0; i < n; ++i) {

q.push(pointer[i]);

}

}

template <typename U, typename... Args>

void construct(U \*p, Args &&... args)

{

new (p) U(std::forward<Args>(args)...);

}

void destroy(pointer p)

{

p->~T();

}

};

template<class T, class ALLOCATOR>

class Myqueue {

public:

template<class S>

struct Node {

using allocator\_type = typename ALLOCATOR::template rebind<Node>::other;

static allocator\_type get\_allocator()

{

static allocator\_type allocator;

return allocator;

}

void \*operator new(size\_t size)

{

return get\_allocator().allocate(sizeof(Node));

}

void operator delete(void \*p)

{

get\_allocator().destroy((Node\*)p);

get\_allocator().deallocate((Node \*)p, sizeof(Node));

}

Node() = default;

Node(const S& elem) : elem(elem) {

next = make\_shared<Node>();

}

~Node() {

next = nullptr;

prev = nullptr;

}

S elem;

shared\_ptr<Node> next;

shared\_ptr<Node> prev;

};

public:

template<class V>class forward\_iterator : public std::iterator<std::forward\_iterator\_tag, V, int> {

public:

friend class Myqueue<T, ALLOCATOR>;

typedef forward\_iterator<V> self\_type;

typedef std::forward\_iterator\_tag iterator\_category;

typedef V value\_type;

typedef V &reference;

typedef weak\_ptr<Node<V> > pointer;

typedef ptrdiff\_t difference\_type;

private:

pointer ptr;

size\_t index = 0;

size\_t size = 0;

bool compatible(self\_type const &other) const {

auto s\_ptr1 = ptr.lock();

auto s\_ptr2 = other.ptr.lock();

return s\_ptr1 == s\_ptr2;

}

public:

explicit forward\_iterator(pointer ptr, size\_t const size, size\_t const index) : ptr(ptr), size(size), index(index) {}

forward\_iterator() = default;

forward\_iterator(forward\_iterator const &o) = default;

forward\_iterator &operator=(forward\_iterator const &o) = default;

~forward\_iterator() = default;

self\_type &operator++() {

if (index >= size) {

throw std::out\_of\_range("Ошибка! Произошел выход за границы очереди!");

}

++index;

auto s\_ptr = ptr.lock();

ptr = s\_ptr->next;

return \*this;

}

self\_type operator++(int) {

self\_type tmp = \*this;

++\*this;

return tmp;

}

bool operator==(self\_type const &other) const {

if (index == other.index) {

assert(compatible(other));

return true;

}

else {

return false;

}

}

bool operator!=(self\_type const &other) const {

return !(\*this == other);

}

reference operator\*() const {

if (index >= size) {

throw std::bad\_function\_call();

}

auto s\_ptr = ptr.lock();

return s\_ptr->elem;

}

reference operator->() const {

if (index >= size) {

throw std::bad\_function\_call();

}

auto s\_ptr = ptr.lock();

return s\_ptr->elem;

}

};

public:

Myqueue() : size(0) {}

~Myqueue() = default;

size\_t Size() { return size; }

auto Begin();

auto End();

void Insert(auto iterator, const T& elem);

void Erase(auto iterator);

T Top();

void Push(const T);

void Pop();

private:

shared\_ptr<Node<T> > begin\_ptr;

shared\_ptr<Node<T> > end\_ptr;

size\_t size;

};

template<class T, class ALLOCATOR>

auto Myqueue<T, ALLOCATOR>::Begin() {

return forward\_iterator<T>(begin\_ptr, size, 0);

}

template<class T, class ALLOCATOR>

auto Myqueue<T, ALLOCATOR>::End() {

return forward\_iterator<T>(end\_ptr, size, size);

}

template<class T, class ALLOCATOR>

void Myqueue<T, ALLOCATOR>::Insert(auto iterator, const T& elem) {

auto new\_elem = make\_shared<Node<T> >(elem);

if (iterator.index == size) {

Push(elem);

}

else if (iterator.index < 0 || iterator.index >= size) {

throw(std::runtime\_error("Ошибка! Выход за границы очереди!"));

}

else if (iterator.index == 0) {

new\_elem->next = begin\_ptr;

begin\_ptr->prev = new\_elem;

begin\_ptr = new\_elem;

++size;

}

else {

auto s\_ptr = iterator.ptr.lock();

s\_ptr->prev->next = new\_elem;

new\_elem->prev = s\_ptr->prev;

s\_ptr->prev = new\_elem;

new\_elem->next = s\_ptr;

++size;

}

}

template<class T, class ALLOCATOR>

void Myqueue<T, ALLOCATOR>::Erase(auto iterator) {

if (iterator.index == 0) {

Pop();

}

else if (iterator.index < 0 || iterator.index >= size) {

throw(std::runtime\_error("Ошибка! Выход за границы очереди!"));

}

else if (iterator.index == size-1) {

end\_ptr->prev->prev->next = end\_ptr;

end\_ptr->prev = end\_ptr->prev->prev;

--size;

}

else {

auto s\_ptr = iterator.ptr.lock();

s\_ptr->next->prev = s\_ptr->prev;

s\_ptr->prev->next = s\_ptr->next;

--size;

}

}

template<class T, class ALLOCATOR>

void Myqueue<T, ALLOCATOR>::Push(const T elem) {

if (size == 0) {

begin\_ptr = make\_shared<Node<T> >(elem);

end\_ptr = begin\_ptr->next;

end\_ptr->prev = begin\_ptr;

}

else {

end\_ptr->elem = elem;

end\_ptr->next = make\_shared<Node<T> >();

end\_ptr->next->prev = end\_ptr;

end\_ptr = end\_ptr->next;

}

++size;

}

template<class T, class ALLOCATOR>

void Myqueue<T, ALLOCATOR>::Pop() {

if (size == 0) {

throw std::runtime\_error("Ошибка! Удаление из пустой очереди!");

}

begin\_ptr->next->prev = nullptr;

begin\_ptr = begin\_ptr->next;

--size;

}

template<class T, class ALLOCATOR>

T Myqueue<T, ALLOCATOR>::Top() {

if (size == 0) {

throw std::runtime\_error("Ошибка! Очередь пуста!");

}

return begin\_ptr->elem;

}

**Содержимое файла main.cpp**

/\* Ляшун Дмитрий Сергеевич, группа М8О-207Б-19 \*/

/\* ЛР №6, Вариант №24: Фигура: 8-угольник, Контейнер: очередь, Контейнер аллокатора: очередь\*/

/\* Назначение программы: работа с коллекцией восьмиугольников: добавление и удаление фигур, их печать, подсчет количества фигур с площадью меньше заданной. \*/

#include "queue.hpp"

#include "octagon.hpp"

#include <algorithm>

using namespace std;

const size\_t ALLOCATOR\_SIZE = 10000;

auto main() -> int {

Myqueue<Octagon<int>, my\_allocator<Octagon<int>, ALLOCATOR\_SIZE> > queue;

cout << "Поддерживаемые операции (для вызова необходимо ввести ее номер): " << endl;

cout << "1 - Добавить восьмиугольник в коллекцию. " << endl;

cout << "2 - Удалить восьмиугольник из коллекции. " << endl;

cout << "3 - Вывод всех восьмиугольников в коллекции. " << endl;

cout << "4 - Выход из программы. " << endl;

while (true) {

int choice;

cout << "> ";

cin >> choice;

if (choice == 1) {

cout << "Введите восьмиугольник (коорд. центра и вектор радиуса описан. окружн.): ";

pair<int,int> vx, vec;

cin >> vx.first >> vx.second >> vec.first >> vec.second;

Octagon<int> oct(vx, vec);

cout << "Введите позицию добавления: ";

int pos;

cin >> pos;

try {

--pos;

auto iter = queue.Begin();

while (pos != 0) {

++iter;

--pos;

}

queue.Insert(iter, oct);

cout << "Восьмиугольник добавлен в коллекцию." << endl;

}

catch (std::exception& exception) {

cout << exception.what() << endl;

}

}

else if (choice == 2) {

cout << "Введите позицию удаления: ";

int pos;

cin >> pos;

try {

--pos;

auto iter = queue.Begin();

while (pos != 0) {

++iter;

--pos;

}

queue.Erase(iter);

cout << "Восьмиугольник удален из коллекции." << endl;

}

catch (std::exception& exception) {

cout << exception.what() << endl;

}

}

else if (choice == 3) {

int count = 0;

for\_each(queue.Begin(),

queue.End(),

[&count](Octagon<int> & iter) {

++count;

cout << "Восьмиугольник №" << count << " : ";

iter.print();

});

}

else if (choice == 4) {

break;

}

else {

cout << "Ошибка! Неправильный ввод!" << endl;

}

}

}

**Содержимое файла CMakeLists.txt**

cmake\_minimum\_required(VERSION 2.8) # Указание необходимой версии CMake

set(SOURCES main.cpp) # присвоение переменной SOURCES значения main.cpp

add\_executable(oop\_exercise\_06 ${SOURCES}) # Компиляция исполняемого файла oop\_exercise\_06

1. **Выводы**

В результате выполнения лабораторной работы я изучил основы работы с контейнерами, а также познакомился с концепцией использования аллокаторов памяти. Основную трудность в ходе работы для меня составило написание собственного аллокатора и его отладка, поскольку по-началу возникали утечки памяти при его использовании.

**Список литературы**

1. Справочник по языку C++ [Электронный ресурс]. URL: [http://www.cplusplus.com/](about:blank) (дата обращения: 02.12.2020).

2. Archived Stack Overflow Documentation [Электронный ресурс]. URL: <https://www.riptutorial.com/> (дата обращения 02.12.2020).