**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 5**

Тема: Основы работы с коллекциями: итераторы

Студент: Шараковский Юрий Дмитриевич

Группа: 80-206

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. Постановка задачи

Создать шаблон динамической коллекцию, согласно варианту задания.

1. Коллекция должна быть реализована с помощью умных указателей.
2. В качестве параметра шаблона коллекция должна принимать тип данных - фигуры.

3. Реализовать forward\_iterator по коллекции.

4. Коллекция должны возвращать итераторы begin() и end().

5. Коллекция должна содержать метод вставки на позицию итератора insert(iterator).

6. Коллекция должна содержать метод удаления из позиции итератора erase(iterator).

7. При выполнении недопустимых операций (например выход зa границы коллекции или удаление несуществующего элемента) необходимо генерировать исключения.

8. Итератор должен быть совместим со стандартными алгоритмами (например, std::count\_if).

9. Коллекция должна содержать метод доступа:

o Динамический массив – доступ к элементу по оператору [].

10. Реализовать программу, которая:

o Позволяет вводить с клавиатуры фигуры и добавлять в коллекцию.

o Позволяет удалять элемент из коллекции по номеру элемента.

o Выводить на экран введенные фигуры с помощью std::for\_each.

o Выводить на экран количество объектов, у которых площадь меньше заданной (с помощью std::count\_if).

Вариант 29: Ромб, Динамический массив

1. Описание программы

Программа реализует набор классов и структур, необходимый для представления геометрической фигуры - Ромба, и шаблонный класс-контейнер для хранения фигур в виде динамического массива, при этом в программе описывается итератор (соответствующий требованиям std::forward\_iterator) данного класса. В главной функции программы считывается пользовательский ввод, создаются фигуры и вставляются в контейнер, также можно удалять фигуры из контейнера по номеру или печатать информацию о них на экран. Заголовочные файлы vector.hpp и rhombus.hpp можно использовать в качестве самостоятельной библиотеки в других программах.

1. Набор тестов

test01.txt - тест проверяет базовую работу программы: создаются пять ромбов и вставляются в динамический массив, после этого они печатаются на экран. Далее, один из них удаляется, и информация снова печатается на экран. С помощью std::count\_if находится число фигур с площадью меньше 4.

1 0

0 0 1 0 0 1

1 0

0 0 2 0 0 2

1 0

0 0 -1 0 0 1

1 3

0 0 -2 0 0 -2

1 4

0 0 1 1 -1 1

3

4 4

2 1

3

4 2

test02.txt - тест показывает то, что программа отбраковывает некорректные входные данные.

1 0

1 2 3 4 5 6

1. Результаты выполнения тестов

test01.txt

1: insert <x>

2: erase <x>

3: print

4: count area <x>

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Rhombus: [-1,0] [0,1] [1,0] [1,0]

Area: 2

Rhombus: [2,0] [0,2] [-2,0] [-2,0]

Area: 8

Rhombus: [1,0] [0,1] [-1,0] [-1,0]

Area: 2

Rhombus: [-2,0] [0,-2] [2,0] [2,0]

Area: 8

Rhombus: [1,1] [-1,1] [-1,-1] [-1,-1]

Area: 4

Total: 2

Rhombus: [-1,0] [0,1] [1,0] [1,0]

Area: 2

Rhombus: [1,0] [0,1] [-1,0] [-1,0]

Area: 2

Rhombus: [-2,0] [0,-2] [2,0] [2,0]

Area: 8

Rhombus: [1,1] [-1,1] [-1,-1] [-1,-1]

Area: 4

Total: 0

test02.txt

1: insert <x>

2: erase <x>

3: print

4: count area <x>

Enter rhombus (center and two adjacent vertices):

Not a rhombus!

1. Листинг программы

main.cpp

#include <algorithm>

#include <iostream>

#include "rhombus.hpp"

#include "vector.hpp"

int main() {

lab::Vector<lab::rhombus<double>> v;

lab::rhombus<double> r;

std::cout << "1: insert <x>\n2: erase <x>\n3: print\n4: count area <x>" << std::endl;

int option = 0;

size\_t x;

while(std::cin >> option) {

switch (option) {

case 1:

std::cin >> x;

std::cout << "Enter rhombus (center and two adjacent vertices):" << std::endl;

std::cin >> r;

v.insert({v, x}, r);

break;

case 2:

std::cin >> x;

v.erase({v, x});

break;

case 3:

std::for\_each(v.begin(), v.end(), [](auto& r) { std::cout << r << "\n\tArea: " << r.getArea() << '\n'; });

break;

case 4:

std::cin >> x;

std::cout << "Total: " << std::count\_if(v.begin(), v.end(), [x](auto& r) { return r.getArea() < x; }) << std::endl;

break;

default:

std::cout << "I don\'t know that command!\n" << std::endl;

}

}

}

rhombus.hpp

#ifndef OOP\_EXERCISE\_05\_RHOMBUS\_HPP

#define OOP\_EXERCISE\_05\_RHOMBUS\_HPP

#include <cmath>

#include <iostream>

namespace lab {

template <typename T>

struct vertex {

T x;

T y;

vertex(T \_x = T(), T \_y = T()) : x(\_x), y(\_y) {};

vertex& operator=(const vertex& v) {

x = v.x;

y = v.y;

return \*this;

}

friend vertex getVector(const vertex& a, const vertex& b) {

return {b.x - a.x, b.y - a.y};

}

friend double dot(const vertex& a, const vertex& b) {

return a.x\*b.x + a.y\*b.y;

}

friend double dist(const vertex& a, const vertex& b) {

return std::sqrt((a.x - b.x)\*(a.x - b.x) + (a.y - b.y)\*(a.y - b.y));

}

friend std::ostream& operator<<(std::ostream& os, const vertex& v) {

os << '[' << v.x << ',' << v.y << ']';

return os;

}

friend std::istream& operator>>(std::istream& is, vertex& v) {

is >> v.x >> v.y;

return is;

}

};

template <typename T>

class rhombus {

public:

friend std::ostream& operator<<(std::ostream& os, const rhombus& r) {

os << "Rhombus: " << r.a << ' ' << r.b << ' ' << r.c << ' ' << r.c;

return os;

}

friend std::istream& operator>>(std::istream& is, rhombus& r) {

vertex<T> x, i, j;

is >> x >> i >> j;

if (!is.good()) {

std::cerr << "Read error!\n";

return is;

}

vertex<T> u(getVector(x, i));

vertex<T> v(getVector(x, j));

vertex<T> k(getVector(u, x));

vertex<T> l(getVector(v, x));

if (dot(getVector(i, k), getVector(j, l)) != 0) {

std::cerr << "Not a rhombus!\n";

return is;

}

r.a = i; r.b = j; r.c = k; r.d = l;

return is;

}

double getArea() {

return dist(a, c) \* dist(b, d) / 2;

}

private:

vertex<T> a, b, c, d;

};

}

#endif //OOP\_EXERCISE\_05\_RHOMBUS\_HPP

vector.hpp

#ifndef OOP\_EXERCISE\_05\_VECTOR\_HPP

#define OOP\_EXERCISE\_05\_VECTOR\_HPP

#include <exception>

#include <stdexcept>

#include <algorithm>

#include <functional>

#include <memory>

namespace lab {

template <typename T>

class Vector {

public:

using index\_type = size\_t;

using value\_type = T;

Vector() : \_length(0), \_capacity(BASE\_CAP) {

\_length = 0;

\_data = std::make\_unique<value\_type[]>(\_capacity);

};

//explicit Vector(index\_type size = 0, const value\_type& value = value\_type());

Vector(const Vector&) = delete;

Vector(Vector&&) = delete;

~Vector() = default;

class VectorIterator {

public:

using difference\_type = Vector::index\_type;

using value\_type = Vector::value\_type;

using reference = Vector::value\_type&;

using pointer = Vector::value\_type\*;

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

VectorIterator(Vector& vector, difference\_type index) : \_vector(vector), \_index(index) {}

VectorIterator& operator++() {

++\_index;

return \*this;

}

reference operator\*() {

return \_vector[\_index];

}

pointer operator->() {

return &\_vector[\_index];

}

bool operator!=(const VectorIterator& other) {

if (\_index != other.\_index) {

return true;

}

if (&\_vector != &(other.\_vector)) {

return true;

}

return false;

}

bool operator==(const VectorIterator& other) {

if (\_index != other.\_index) {

return false;

}

if (&\_vector != &(other.\_vector)) {

return false;

}

return true;

}

private:

Vector& \_vector;

difference\_type \_index;

friend class Vector;

};

index\_type& length() {

return \_length;

};

[[nodiscard]] const index\_type& length() const {

return \_length;

};

value\_type& at(index\_type index) {

if (index >= 0 && index < \_capacity) {

return \_data[index];

}

else {

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

}

};

const value\_type& at(index\_type index) const {

if (index >= 0 && index < \_capacity) {

return \_data[index];

}

else {

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

}

};

value\_type& operator[](index\_type index) noexcept {

return \_data[index];

};

const value\_type& operator[](index\_type index) const noexcept {

return \_data[index];

};

index\_type size() {

return \_length;

}

VectorIterator begin() {

return VectorIterator(\*this, 0);

}

VectorIterator end() {

return VectorIterator(\*this, size());

}

void reserve(index\_type capacity) {

if (capacity < \_capacity) {

return;

}

else {

while (capacity < \_capacity) {

index\_type new\_capacity = capacity \* Vector::GROWTH\_RATE;

if (new\_capacity > capacity) {

capacity = new\_capacity;

}

else {

break;

}

}

auto old\_data = std::move(\_data);

try {

\_data = std::make\_unique<value\_type[]>(capacity);

}

catch (std::bad\_alloc& ex) {

std::cerr << "ERROR: " << ex.what() << std::endl;

}

for (index\_type index = 0; index < \_capacity; ++index) {

\_data[index] = old\_data[index];

}

\_capacity = capacity;

}

}

void push\_back(value\_type& value) {

if (\_length >= \_capacity) {

reserve(\_capacity + 1);

}

if (\_length >= \_capacity) {

throw std::logic\_error("max capacity reached");

}

\_data[\_length] = value;

++\_length;

}

void pop\_back(value\_type& value) {

if (\_length > 0) {

--\_length;

value = \_data[\_length];

}

else {

throw std::logic\_error("pop on empty vector");

}

}

VectorIterator erase(VectorIterator iter) {

if (iter.\_index < 0 || iter.\_index > size()) {

throw std::logic\_error("bad insert");

}

index\_type i = 0;

for (; i < \_length; ++i) {

if (i == iter.\_index) {

break;

}

}

for (; i < \_length; ++i) {

\_data[i] = \_data[i + 1];

}

--\_length;

return iter;

}

VectorIterator insert(VectorIterator iter, const value\_type& value) {

if (iter.\_index < 0 || iter.\_index > size()) {

throw std::logic\_error("bad insert");

}

if (\_length == \_capacity) {

reserve(\_length + 1);

}

for (index\_type i = \_length; i > 0; --i) {

\_data[i] = \_data[i - 1];

if (i == iter.\_index) {

\_data[i] = value;

break;

}

}

if (iter.\_index == 0) {

\_data[0] = value;

}

++\_length;

return iter;

}

private:

static index\_type const BASE\_CAP = 10;

static index\_type const GROWTH\_RATE = 2;

std::unique\_ptr<value\_type[]> \_data;

index\_type \_length;

index\_type \_capacity;

};

}

#endif //OOP\_EXERCISE\_05\_VECTOR\_HPP

1. Вывод

Механизм итераторов позволяет нам иметь одинаковый подход к работе с многими коллекциями, а также придать пользовательским коллекциям дополнительную гибкость, благодаря им мы можем не только легко итерироваться по всем элементам некоторой коллекции, но и использовать их в качестве параметров для многих функций стандартной библиотеки, работающих с коллекциями - инспекторов, мутаторов и фасилитаторов (например std::for\_each, std::find, std::find\_if, std::transform, std::count\_if и.т.д). Описав свой итератор для класса, мы можем пользоваться всеми этими функциями.

Список литературы

1. C++ Primer (5th Edition) Stanley B. Lippman, Josee Lajoie, E. Moo. – Addison-Wesley, 2019. ISBN-13: 978-0321714114, ISBN-10: 9780321714114.

2. Effective Modern C++, Scott Meyers. ISBN-13: 978-1491903995

ISBN-10: 9781491903995