Московский авиационный институт

(Национальный исследовательский университет)

Факультет: «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: Кудинов Сергей

Преподаватель: Журавлев А.А.

Дата:

Оценка:

Москва, 2019

1. **Постановка задачи**

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных, задающий тип данных для оси координат. Создать шаблон динамической коллекции согласно варианту задания, в соответствии со следующими требованиями:

- Коллекция должна быть реализована с помощью умных указателей
- В качестве шаблона коллекция должна принимать тип данных.
- Реализовать однонаправленный итератор по коллекции.
- Коллекция должна возвращать итераторы на начало и конец.
- Коллекция должна содержать метод вставки на позицию итератора.
- Коллекция должна содержать метод удаления из позиции итератора.
- При выполнении недопустимых операций(выход за границы коллекции или удаление несуществующего элемента) необходимо генерировать исключения.
- Итератор должен быть совместим со стандартными алгоритмами.
- Коллекция должна содержать метод доступа – pop, push, top.
- Реализовать программу, которая позволяет вводить с клавиатуры фигуры, удалять элемент из коллекции по номеру, выводит выведенные фигуры с помощью for_each, выводит на экран количество элементов, у которых площадь меньше заданной.

Вариант задания 11:

- Фигура - Прямоугольник
- Коллекция - список

1. **Репозиторий github**

   https://github.com/StormStudioAndroid2/oop_exercise_05/

2. **Описание программы**

Реализован шаблонный класс списка. Данные хранятся с помощью unique_pt. Также реализованы класс для итератора. Лист содержит два барьерных элемента для упрощения функций вставки, удаления и итерирования. Коллекция может также работать со стандартными алгоритмами.

3. **Набор testcases**

push

1 0 0 1 0 0 1 1

all

push

5 0 0 0 5 5 0 5

all

delete

0

all

4. **Результаты выполнения тестов**
5. Rectangle p1: 0 1p1 0 0p2 1 0p3 1 1p4
6.
7. Rectangle p1: 0 1p1 0 0p2 1 0p3 1 1p4
8. Rectangle p1: 5 5p1 5 0p2 0 0p3 0 5p4
9.
10.
11. Rectangle p1: 5 5p1 5 0p2 0 0p3 0 5p4
12. **Листинг программы**

main.cpp

```
#include <iostream>
#include "Rectangle.h"
#include "./containers/list.h"
#include <string.h>
#include <algorithm>

int main() {
    char str[10];
    containers::list<Rectangle<double> > l;
    auto it = l.begin();
    while(std::cin >> str){
        if(strcmp(str,"push")==0){
            Rectangle<double> rectangle;
            rectangle.scan(std::cin);
```

```
    l.add(rectangle);
}else if(strcmp(str,"delete")==0){
    try {
        int t;
        std::cin >> t;

        std::next(it,t);
        l.erase(l.begin()+t);
        std::cout << "\n";
    }catch (std::exception& ex){
        std::cout <<ex.what() << "\n";
    }
}
else if(strcmp(str,"front")==0) {
    try {
       l.front().print(std::cout);
        std::cout << "\n";
    }catch (std::exception& ex){
        std::cout <<ex.what() << "\n";
    }
}if(strcmp(str,"get")==0) {
    try {
       int r;
       std::cin >> r;
        l[r].print(std::cout);
        std::cout << "\n";
    }catch (std::exception& ex){
        std::cout <<ex.what() << "\n";
    }
} else if(strcmp(str,"end")==0){
    try {

        l.End().print(std::cout);
    }catch (std::exception& ex){
        std::cout <<ex.what() << "\n";
    }
}else if(strcmp(str,"square")==0) {
    int g;
    std::cin >> g;
    long res=std::count_if(l.begin(),l.end(),[g](Rectangle<double> f){ return f.getSquare() < g;});
    std::cout << res << "\n";
} else if(strcmp(str,"insert")==0){
    int r;
    std::cin >>r;
    Rectangle<double> rectangle;
    rectangle.scan(std::cin);
    l.insert(l.begin() + r,rectangle);
}else if(strcmp(str,"all")==0){
    if (l.begin()!=nullptr) {
    std::for_each(l.begin(),l.end(),[](Rectangle<double> f){f.print(std::cout); });
    std::cout<< "\n";
    } else {
```

```
            std::cout << "Empty list!" << std::endl;
          }
       }

   }
   return 0;
}
```

Rectangle.h

```
#pragma once
#include "Point.h"
#include <vector>
template <typename T>

class Rectangle {
   public:
   Point<T> points[4];
   Rectangle<T>() = default;
   Rectangle(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);
   double getSquare() const;
   Point<T> getCenter() const;
   void scan(std::istream &is);
   void print(std::ostream& os) const;
};



template <typename T>
Rectangle<T>::Rectangle(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4) {
      if (IsRectangle(p1,p2,p3,p4)) {

   } else if (IsRectangle(p2, p3, p1, p4)) {
      std::swap(p2, p1); std::swap(p3,p2);
   } else if (IsRectangle(p3, p1, p2, p4)) {
      std::swap(p3, p1); std::swap(p3,p2);
   } else {
      throw std::logic_error("not rectangle");
   }
   this->points[0] = p1;
   this->points[1] = p2;
   this->points[2] = p3;
   this->points[3] = p4;
   }

   template <typename T>
double Rectangle<T>::getSquare() const {

          return length(this->points[0],this->points[1])*length(this->points[0],this->points[3]);
}
template <typename T>
void Rectangle<T>::print(std::ostream& os) const {
   os << "Rectangle p1: ";
   for (int i = 0; i < 4; ++i) {
```

```
        os << this->points[i] << "p" << i+1 <<" ";
    }
    os << std::endl;
}
template <typename T>
void Rectangle<T>::scan(std::istream &is) {
    Point<T> p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Rectangle(p1,p2,p3,p4);
}

template <typename T>
Point<T> Rectangle<T>::getCenter() const {

    Point<T> p;
    p.x = 0;
    p.y = 0;
    for (size_t i = 0; i < 4; ++i) {
        p = p+(points[i]/4);
    }
    return p;
}
```

list.h

```
#ifndef D_LIST_H_
#define D_LIST_H_

#include <iostream>
#include <memory>
#include <functional>
#include <cassert>
#include <iterator>


namespace containers {

    template<class T>
    struct list {
    private:
        struct node;

    public:
        list() = default;
        T End();
        T& operator[] (const int index);
        size_t Size();
        struct forward_iterator {
            using value_type = T;
            using reference = T &;
            using pointer = T *;
            using difference_type = ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;
```

```
        forward_iterator(node *ptr);

        T &operator*();

        forward_iterator &operator++();

        forward_iterator operator+(int r);

        bool operator==(const forward_iterator &o) const;

        bool operator!=(const forward_iterator &o) const;

    private:
        node *ptr_;

        friend list;

    };

    forward_iterator begin();

    forward_iterator end();

    void insert(const forward_iterator &it, const T &value);

    void erase(const forward_iterator &it);

    void popStart();
    void popEnd();

    void add(const T &value);

    T front();

private:
    node *end_node = nullptr;

    node *getTail(node *ptr);

    struct node {
        T value;
        std::unique_ptr<node> next = nullptr;
        node *parent = nullptr;

        forward_iterator nextf();
    };

    std::unique_ptr<node> root = nullptr;
    node* rootEnd = nullptr;

};
```

```
//
template<class T>
size_t list<T>::Size() {
    auto it = begin();
    size_t size1 = 0;
    while(it!=0) {
        ++it;
        size1++;
    }
    return size1;
}
    template<class T>
    typename list<T>::node *list<T>::getTail(containers::list<T>::node *ptr) {
        if ((ptr == nullptr) || (ptr->next == nullptr)) {
            return ptr;
        }
        return list<T>::getTail(ptr->next.get());
    }


    template<class T>
    typename list<T>::forward_iterator list<T>::begin() {
        if (root == nullptr) {
            return nullptr;
        }
        forward_iterator it(root.get());
        return it;
    }

    template<class T>
    typename list<T>::forward_iterator list<T>::end() {
        return nullptr;
    }

    template<class T>
    void list<T>::insert(const list<T>::forward_iterator &it, const T &value) {
        std::unique_ptr<node> new_node(new node{value});
        if (it != nullptr) {
            node *ptr = it.ptr_->parent;
            new_node->parent = it.ptr_->parent;
            it.ptr_->parent = new_node.get();
            if (ptr) {
                new_node->next = std::move(ptr->next);
                ptr->next = std::move(new_node);
            } else {
                new_node->next = std::move(root);
                root = std::move(new_node);
            }
        } else {
            new_node->next = nullptr;
```

```
            if(end_node==nullptr) {
                new_node->parent= nullptr;
                new_node->next= nullptr;
                list<T>::root = std::move(new_node);
            }else{
                new_node->parent=end_node;
                new_node->next= nullptr;
                end_node->next=std::move(new_node);
            }
        }

        end_node = getTail(root.get());
    }
    template<class T>
    T list<T>::End() {
        return end_node->value;
    }
    template<class T>
    void list<T>::erase(const list<T>::forward_iterator &it) {
        if (it.ptr_ == nullptr) {
            throw std::logic_error("erasing invalid iterator");
        }
        std::unique_ptr<node> &pointer_from_parent = [&]() -> std::unique_ptr<node> & {
            if (it.ptr_ == root.get()) {
                return root;
            }
            return it.ptr_->parent->next;
        }();
        pointer_from_parent = std::move(it.ptr_->next);

        end_node = getTail(root.get());
    }

//
    template<class T>
    typename list<T>::forward_iterator list<T>::node::nextf() {
        forward_iterator result(this->next.get());
        return result;
    }
template<class T>
T&  list<T>::operator[] (const int index) {
    if (index>Size()) {
            throw std::logic_error("invalid index");

    }
    auto it = this->begin()+index;
    return *it;
}

    template<class T>
    list<T>::forward_iterator::forward_iterator(node *ptr): ptr_{ptr} {}
```

```
template<class T>
T &list<T>::forward_iterator::operator*() {
    return ptr_->value;
}

template<class T>
typename list<T>::forward_iterator &list<T>::forward_iterator::operator++() {
    if (*this != nullptr) {
        *this = ptr_->nextf();
        return *this;
    } else {
        throw std::logic_error("invalid iterator");
    }
}

template<class T>
typename list<T>::forward_iterator list<T>::forward_iterator::operator+(int r) {
    for (int i = 0; i < r; ++i) {
        ++*this;
    }
    return *this;
}

template<class T>
bool list<T>::forward_iterator::operator==(const forward_iterator &o) const {
    return ptr_ == o.ptr_;
}

template<class T>
bool list<T>::forward_iterator::operator!=(const forward_iterator &o) const {
    return ptr_ != o.ptr_;
}

template<class T>
T list<T>::front() {
    if (list<T>::root == nullptr) {
        throw std::logic_error("no elements");
    }
    return list<T>::root->value;
}

template<class T>
void list<T>::popStart() {
    if (list<T>::root == nullptr) {
        throw std::logic_error("no elements");
    }
    erase(list<T>::begin());
}
    template<class T>
void list<T>::popEnd() {
    if (list<T>::root == nullptr) {
        throw std::logic_error("no elements");
```

```
        }
        erase(list<T>::getTail(list<T>::begin()));
    }
    template<class T>
    void list<T>::add(const T &value) {
        forward_iterator it(end_node);
        std::unique_ptr<node> new_node(new node{value});
        if (it.ptr_) {
            new_node->parent = it.ptr_;
            it.ptr_->next = std::move(new_node);
        } else {
            new_node->next = nullptr;
            list<T>::root = std::move(new_node);
        }
        list<T>::end_node = getTail(root.get());
    }

}
#endif
```

Point.h

```
#pragma once

#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>

template <typename T>
struct Point {
    T x;
    T y;
};
template <typename T>
Point<T> operator+(Point<T> left, Point<T> right) {
     Point<T> p;
    p.x = left.x+right.x;
    p.y = left.y+right.y;
        return p;

}

template <typename T>
 Point<T> operator+( Point<T> left, int right) {
     Point<T> p;
    p.x = left.x+right;
    p.y = left.y+right;
        return p;

}
```

```
template <typename T>
 Point<T> operator-( Point<T> left,  Point<T> right) {
     Point<T> p;
    p.x = left.x-right.x;
    p.y = left.y-right.y;
        return p;

}

template <typename T>
 Point<T> operator-(Point<T> left, double right) {
     Point<T> p;
    p.x = left.x-right;
    p.y = left.y-right;
        return p;

}

template <typename T>
 Point<T> operator/( Point<T> left, double right) {
    Point<T> p;
    p.x = left.x/right;
    p.y = left.y/right;
        return p;

}

template <typename T>
 Point<T> operator*(Point<T> left, double right) {
     Point<T> p;
    p.x = left.x*right;
    p.y = left.y*right;
    return p;
}
template <typename T>
double length( Point<T> left,  Point<T> right) {
    return sqrt((left.x-right.x)*(left.x-right.x)+(left.y-right.y)*(left.y-right.y));
}
template <typename T>
bool IsOrthogonal( Point<T> a,  Point<T> b,  Point<T> c)
{
    return (b.x - a.x) * (b.x - c.x) + (b.y - a.y) * (b.y - c.y) == 0;
}
template <typename T>

bool IsParallel( Point<T> a,  Point<T> b,  Point<T> c,  Point<T> d)
{
     Point<T> a1 = a-b;
     Point<T> a2 = c-d;
    return                                  ((a1.x*a2.x+a1.y*a2.y)/(length(a,b)*length(c,d))<=-1                                  ||
(a1.x*a2.x+a1.y*a2.y)/(length(a,b)*length(c,d))>=1);
}
```

```
template <typename T>
int IsRectangle( Point<T> a,  Point<T> b,  Point<T> c,  Point<T> d)
{
   return
      IsOrthogonal(a, b, c) &&
      IsOrthogonal(b, c, d) &&
      IsOrthogonal(c, d, a);
}

template <typename T>
std::ostream& operator << (std::ostream& os, const Point<T>& p) {
   return os << p.x << " " << p.y;
}

template <typename T>
std::istream& operator >> (std::istream& is, Point<T>& p) {
   return is >> p.x >> p.y;
}
```

13. Вывод

В результате данной работы я получил навыки реализации шаблонных контейнеров, а так же научился работать с умными указателями и создавать свои итераторы.