

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №7

по курсу объектно-ориентированное программирование I семестр, 2021/22 уч.
год

Студент Абдуль-Хади Филипп, группа М8О-207Б-21

Преподаватель Дорохов Евгений Павлович

Условие

Задание

Используя структуру данных, разработанную для лабораторной работы No4, спроектировать и

разработать итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми

типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа `for`.

Например:

```
for(auto i : stack) {  
std::cout << *i << std::endl;  
}
```

Нельзя использовать:

- Стандартные контейнеры `std`.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер;
- Распечатывать содержимое контейнера;
- Удалять фигуры из контейнера.

Описание программы

Исходный код лежит в 9 файлах:

1. `main.cpp`: основная программа, взаимодействие с пользователем посредством ввода команд
2. `figure.h`: класс фигуры
3. `figure.cpp`: реализация класса фигуры
4. `point.h`: класс фигуры
5. `point.cpp`: реализация класса фигуры
6. `square.h`: класс квадрата
7. `square.cpp`: реализация класса квадрата
8. `tvector.h`: класс вектора
9. `tvector.cpp`: реализация вектора
10. `titerator.h`: класс итератора

Дневник отладки

Проблема

Неудобно собирать без make файла

Исправление

Создал make файл

Недочёты

Отсутствуют

Выводы

Итераторы позволяют удобно перебирать различные структуры данных и работать с их элементами, таким образом итераторы являются полезной возможностью языка. Отметим что не всё можно перебрать обычным циклом `for` с счетчиком индекса, например при переборе элементов связанного списка мы должны использовать указатель на текущую ноду, и заменять его на `next` этой ноды. Итераторы берут на себя реализацию перебора подобных коллекций.

Исходный код ниже:

//MAIN.CPP

```
#include <iostream>
#include "square.h"
#include "tvector.h"
#include "tvector.cpp"

using namespace std;

int main()
{
    cout << "Comands:" << endl;
    cout << "a - add new square (a [input])" << endl;
    cout << "d - erase square by index (d [idx])" << endl;
    cout << "s - set square by index (s [idx] [input])" << endl;
    cout << "p - print all containing squares (p)" << endl;
    cout << "q - quit (q)" << endl;
    char running = 1;
    TVector<Figure> *vect = new TVector<Figure>();
    char cmd;
    while(running)
    {
        cout << "> ";
        cin >> cmd;
        switch(cmd)
        {
            case 'a':
            {
                vect->InsertLast(shared_ptr<Figure>(new
Square(cin)));
                break;
            }
            case 'd':
            {
                int di;
                cin >> di;
                vect->Erase(di);
                break;
            }
            case 's':
            {
                int si;
                cin >> si;
                (*vect)[si] = shared_ptr<Figure>(new
Square(cin));
                break;
            }
            case 'p':
            {
                for(Figure *elem: *vect)
                {
                    cout<<*elem<<" "<<endl;
                }
                break;
            }
            case 'q':
            {
                running = 0;
            }
        }
    }
}
```

```
                break;
            }
            default:
                cout << "wrong input" << endl;
        }
    }
    delete vect;
}
```

```
//POINT.H
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
    Point();
    Point(std::istream &is);
    Point(double x, double y);

    double x_;
    double y_;

    double dist(Point& other);

    Point& operator=(const Point& sq);
    bool operator==(const Point& sq);

    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);
};

#endif // POINT_H
```

```

//POINT.CPP
#include "point.h"

#include <cmath>

Point::Point() : x_(0.0), y_(0.0) {}

Point::Point(double x, double y) : x_(x), y_(y) {}

Point::Point(std::istream &is) {
    is >> x_ >> y_;
}

double Point::dist(Point& other) {
    double dx = (other.x_ - x_);
    double dy = (other.y_ - y_);
    return std::sqrt(dx*dx + dy*dy);
}

Point& Point::operator=(const Point& sq)
{
    x_ = sq.x_;
    y_ = sq.y_;
    return *this;
}

bool Point::operator==(const Point& sq)
{
    return x_==sq.x_ && y_==sq.y_;
}

std::istream& operator>>(std::istream& is, Point& p) {
    is >> p.x_ >> p.y_;
    return is;
}

std::ostream& operator<<(std::ostream& os, Point& p) {
    os << "(" << p.x_ << ", " << p.y_ << ")";
    return os;
}

```

```

//FIGURE.H
#ifndef FIGURE_H
#define FIGURE_H

#include <iostream>
#include "point.h"

class Figure
{
    public:
        virtual size_t VertexesNumber() = 0;
        virtual double Area() = 0;
        virtual void Print(std::ostream& os) = 0;
        virtual void Read(std::istream& is) = 0;
        double calcTriangleArea(Point p1, Point p2, Point p3);
        virtual ~Figure() {}

        friend std::ostream& operator<<(std::ostream& os, Figure& p)
        {
            p.Print(os);
            return os;
        }

        friend std::istream& operator>>(std::istream& is, Figure& p)
        {
            p.Read(is);
            return is;
        }
};

#endif // FIGURE_H

```



```
//FIGURE.CPP
#include "figure.h"

#include <cmath>

double Figure::calcTriangleArea(Point p1, Point p2, Point p3)
{
    return abs((p1.x_-p3.x_)*(p2.y_-p3.y_)-(p2.x_-p3.x_)*(p1.y_-
p3.y_))/2.0;
}
```

```

//SQUARE.H
#ifndef SQUARE_H
#define SQUARE_H

#include <iostream>
#include "figure.h"

class Square: public Figure
{
    public:
        Square(std::istream &is);
        Square(Point pnt1, Point pnt2, Point pnt3, Point pnt4);

        size_t VertexesNumber();
        double Area();
        void Print(std::ostream& os);
        void Read(std::istream& is);

        Square& operator=(const Square& sq);
        bool operator==(const Square& sq);

    friend std::istream& operator>>(std::istream& is, Square& r);
    friend std::ostream& operator<<(std::ostream& os, Square& r);

    private:
        Point p1;
        Point p2;
        Point p3;
        Point p4;
};

#endif

```

```

//SQUARE.CPP
#include "square.h"

#include <cmath>

Square::Square(std::istream &is)
{
    is >> *this;
}

Square::Square(Point pnt1, Point pnt2, Point pnt3, Point pnt4)
{
    p1 = pnt1;
    p2 = pnt2;
    p3 = pnt3;
    p4 = pnt4;
}

size_t Square::VertexesNumber()
{
    return 4;
}

double Square::Area()
{
    return calcTriangleArea(p1,p2,p3)+calcTriangleArea(p3,p4,p1);
}

void Square::Print(std::ostream& os)
{
    os << *this;
}

void Square::Read(std::istream& is)
{
    is >> *this;
}

Square& Square::operator=(const Square& sq)
{
    p1=sq.p1;
    p2=sq.p2;
    p3=sq.p3;
    p4=sq.p4;
    return *this;
}

bool Square::operator==(const Square& sq)
{
    return p1==sq.p1 && p2==sq.p2 && p3==sq.p3 && p4==sq.p4;
}

std::istream& operator>>(std::istream& is, Square& r) {
    is >> r.p1 >> r.p2 >> r.p3 >> r.p4;
    return is;
}

std::ostream& operator<<(std::ostream& os, Square& r) {
    os << "Square: " << r.p1 << " " << r.p2 << " " << r.p3 << " " << r.p4;
}

```

```
    return os;  
}
```

```

//TVCETOR.H

#ifndef TVECTOR_H
#define TVECTOR_H

#include <iostream>
#include "square.h"
#include "titerator.h"
#include <memory>

template<class E>
class TVector
{
    public:
        TVector();
        TVector(const TVector& other);
        void Erase(int pos);
        void InsertLast(const std::shared_ptr<E>& elem);
        void RemoveLast();
        const std::shared_ptr<E>& Last();
        std::shared_ptr<E>& operator[](const size_t idx);
        bool Empty();
        size_t Length();
        void Clear();
        TIterator<E> begin();
        TIterator<E> end();
        ~TVector();
    private:
        void resize(int newsize);
        std::shared_ptr<E> *vals;
        int len;
        int rLen;
};

#endif

```

```

//TVECTOR.CPP
#include "tvector.h"
#include <iostream>
#include <cstring>

template<class E>
TVector<E>::TVector()
{
    vals = NULL;
    len = 0;
    rLen = 0;
}

template<class E>
TVector<E>::TVector(const TVector& other)
{
    len = other.len;
    rLen = other.rLen;
    vals = (std::shared_ptr<E>*)malloc(sizeof(std::shared_ptr<E>)*len);
    memcpy((void*)vals, (void*)other.vals,
sizeof(std::shared_ptr<E>)*len);
}

template<class E>
void TVector<E>::Erase(int pos)
{
    if(len == 1)
    {
        Clear();
        return;
    }
    vals[pos] = NULL;
    memmove((void*)&(vals[pos]),
(void*)&(vals[pos+1]), sizeof(std::shared_ptr<E>)*(len-pos-1));
    len--;
    if(len==rLen>>1)
        resize(len);
}

template<class E>
void TVector<E>::InsertLast(const std::shared_ptr<E>& elem)
{
    if(rLen)
    {
        if(len>=rLen)
        {
            rLen<=1;
            resize(rLen);
        }
    }
    else
    {
        rLen=1;
        resize(rLen);
    }
    vals[len] = elem;
    len++;
}

```

```

template<class E>
void TVector<E>::RemoveLast()
{
    Erase(len-1);
}

template<class E>
const std::shared_ptr<E>& TVector<E>::Last()
{
    return vals[len-1];
}

template<class E>
std::shared_ptr<E>& TVector<E>::operator[](const size_t idx)
{
    return vals[idx];
}

template<class E>
bool TVector<E>::Empty()
{
    return len == 0;
}

template<class E>
size_t TVector<E>::Length()
{
    return len;
}

template<class E>
std::ostream& operator<<(std::ostream& os, TVector<E>& obj)
{
    os << '[';
    for(size_t i = 0; i < obj.Length(); i++)
    {
        os << obj[i].get()->Area();
        if(i != obj.Length() - 1)
            os << " ";
    }
    os << ']';
    return os;
}

template<class E>
void TVector<E>::Clear()
{
    if(!Empty())
    {
        for(int i=0;i<len;i++)
            vals[i]=NULL;
        free(vals);
        vals = NULL;
        len = 0;
        rLen = 0;
    }
}

template<class E>

```

```
void TVector<E>::resize(int newsize)
{
    vals = (std::shared_ptr<E>*)realloc((void*)vals,
sizeof(std::shared_ptr<E>)*newsize);
}

template<class E>
TIterator<E> TVector<E>::begin()
{
    return TIterator<E>(vals, 0);
}

template<class E>
TIterator<E> TVector<E>::end()
{
    return TIterator<E>(vals, len);
}

template<class E>
TVector<E>::~~TVector()
{
    Clear();
}
```



```

//TITERATOR.H

#ifndef TITERATOR_H
#define TITERATOR_H

#include <memory>

template<class E> class TIterator
{
    public:
        TIterator<E>(std::shared_ptr<E> *vct, int ix)
        {
            vect = vct;
            idx = ix;
        }

        E* operator*()
        {
            return vect[idx].get();
        }

        E* operator->()
        {
            return vect[idx].get();
        }

        void operator++()
        {
            idx++;
        }

        TIterator<E> operator++(int)
        {
            TIterator iter(vect,idx);
            ++(*this);
            return iter;
        }

        bool operator==(TIterator const& i)
        {
            return idx == i.idx;
        }

        bool operator!=(TIterator const& i)
        {
            return idx != i.idx;
        }

    private:
        std::shared_ptr<E> *vect;
        int idx;
};

#endif

```