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 *Борисов Ян Артурович, группа М80-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

### Цель работы

Целью лабораторной работы является:

Закрепление навыков работы с шаблонами классов;

Построение итераторов для динамических структур данных.

### Задание

Используя структуру данных, разработанную для лабораторной работы №4, 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.

**Дневник отладки**

Во время выполнения лабораторной работы были некие неисправности в итерировании по контейнеру в силу нелинейности бинарного дерева. В финальном варианте программы все работает исправно.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №7 позволила мне реализовать свой класс Iterator на языке С++, были освоены базовые навыки работы с самописными итераторами и итерирование по созданному контейнеру.

**Исходный код**

Titerator.h

#pragma once

#include <memory>

template <typename T>

class TIterator {

public:

inline TIterator(std::shared\_ptr<T> \*iterator)

: iter\_(iterator) {}

inline T operator\*() const

{

return \*(\*iter\_);

}

inline T operator->() const

{

return \*(\*iter\_);

}

inline void operator++()

{

iter\_ += 1;

}

inline TIterator operator++(int)

{

TIterator iter(\*this);

++(\*this);

return iter;

}

inline bool operator==(TIterator const &iterator) const

{

return iter\_ == iterator.iter\_;

}

inline bool operator!=(TIterator const &iterator) const

{

return !(\*this == iterator);

}

private:

std::shared\_ptr<T> \*iter\_;

};

TVector.h

#pragma once

#include <ostream>

#include <memory>

#include "titerator.h"

template <typename T>

class TVector {

public:

TVector();

TVector(const TVector &);

virtual ~TVector();

inline size\_t Length() const

{

return length\_;

}

inline bool Empty() const

{

return !length\_;

}

inline const std::shared\_ptr<T> &operator[](const size\_t index) const

{

return data\_[index];

}

inline std::shared\_ptr<T> Last() const

{

return data\_[length\_ - 1];

}

void InsertLast(const std::shared\_ptr<T> &);

void EmplaceLast(const T &&);

void Remove(const size\_t index);

inline T RemoveLast()

{

return \*data\_[--length\_];

}

void Clear();

inline TIterator<T> begin()

{

return TIterator<T>(data\_);

}

inline TIterator<T> end()

{

return TIterator<T>(data\_ + length\_);

}

template <typename TF> friend std::ostream &operator<<(

std::ostream &, const TVector<TF> &);

private:

void \_Resize(const size\_t new\_capacity);

std::shared\_ptr<T> \*data\_;

size\_t length\_, capacity\_;

enum { CAPACITY = 32 };

};

#include <cstdlib>

template <typename T>

TVector<T>::TVector()

: data\_(new std::shared\_ptr<T>[CAPACITY]),

length\_(0), capacity\_(CAPACITY) {}

template <typename T>

TVector<T>::TVector(const TVector &vector)

: data\_(new std::shared\_ptr<T>[vector.capacity\_]),

length\_(vector.length\_), capacity\_(vector.capacity\_)

{

std::copy(vector.data\_, vector.data\_ + vector.length\_, data\_);

}

template <typename T>

TVector<T>::~TVector()

{

delete[] data\_;

}

// NOTE: C++ has no `realloc`, so this is a workaround:

template <typename T>

void TVector<T>::\_Resize(const size\_t new\_capacity)

{

std::shared\_ptr<T> \*newdata = new std::shared\_ptr<T>[new\_capacity];

std::copy(data\_, data\_ + capacity\_, newdata);

delete[] data\_;

data\_ = newdata;

capacity\_ = new\_capacity;

}

#define \_EXTEND\_VECTOR \

if (length\_ >= capacity\_) \

\_Resize(capacity\_ << 1);

template <typename T>

void TVector<T>::InsertLast(const std::shared\_ptr<T> &item)

{

\_EXTEND\_VECTOR

data\_[length\_++] = item;

}

template <typename T>

void TVector<T>::EmplaceLast(const T &&item)

{

\_EXTEND\_VECTOR

data\_[length\_++] = std::make\_shared<T>(item);

}

#undef \_EXTEND\_VECTOR

template <typename T>

void TVector<T>::Remove(const size\_t index)

{

std::copy(data\_ + index + 1, data\_ + length\_, data\_ + index);

--length\_;

}

template <typename T>

void TVector<T>::Clear()

{

delete[] data\_;

data\_ = new std::shared\_ptr<T>[CAPACITY];

length\_ = 0;

capacity\_ = CAPACITY;

}

template <typename T>

std::ostream &operator<<(std::ostream &os, const TVector<T> &vector)

{

const size\_t last = vector.length\_ - 1;

for (size\_t i = 0; i < vector.length\_; ++i)

os << \*vector.data\_[i] << ((i != last) ? '**\n**' : '**\0**');

return os;

}