

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 C++, были освоены базовые навыки работы с самописными итераторами и итерирование по созданному контейнеру.

## Исходный код

Iterator.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;
}

```