МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Объектно-ориентированное программирование»

Тема: Конструирование объектов

Студент гр. 7382 _____ Филиппов И.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Знакомство с объектно-ориентированной стороной языка C++.

Основные теоретические положения.

Добавьте в класс Array реализацию конструктора копирования, конструктора перемещения (move constructor), оператора присваивания и, если это необходимо, деструктора. Семантику реализованных функций: конструктор по умолчанию, функция size() и operator [] нужно оставить без изменений. Для хранения данных в классе Array необходимо использовать динамический массив, однако для управления освобождением памяти можно использовать умный указатель (std::unique_ptr).

Разработанный класс должен давать строгую гарантию безопасности исключений (реализовывать семантику *commit-or-rollback*). Это означает, что при сбое операции гарантируется неизменность состояния задействованных в операции объектов. Другими словами, любые функции класса T (за исключением деструкторов) могут генерировать исключения. Объекты такого класса T могут храниться в массиве Array. При этом конструкторы класса Array не могут задерживать сгенерированные объектом класса T исключения, однако должны гарантировать корректность своего состояния в момент бросания исключения.

Требования к реализации: при выполнении этого задания вы можете определять любые вспомогательные функции. Вводить или выводить что-либо не нужно. Реализовывать функцию main не нужно. Не используйте функции из cstdlib (malloc, calloc, realloc и free).

Выводы.

В ходе выполнения лабораторной работы были реализованны конструкторы копирования и перемещения, оператор присваивания и

деструктор класс Array. Оператор присаивания гарантирует строгую гарантию безопасности исключений (commit-or-rollback).

ПРИЛОЖЕНИЕ

Исходный код

```
#pragma once

#include <assert.h>
#include <algorithm>
#include <cstddef>
#include <iostream>

template<typename T>
class Array
{
public:
   explicit Array(const size_t size = 0)
         : m_array_(size ? new T[size]() : nullptr)
         , m_size_(size)
   {}

   Array(const Array& other)
         : m_array_(other.size() ? new T[other.size()]() : nullptr)
         , m_size_(other.size())
   {

      std::copy(other.m_array_, other.m_array_ + other.size(), m_array_);
   }

   Array(Array&& other) noexcept
      : m_array_(nullptr)
      , m_size_(0)
   {
      std::swap(m_array_, other.m_array_);
      std::swap(m_size_, other.m_size_);
   }

   Array& operator=(const Array& other)
   {
      if (this != &other)
      {
         auto m_array_saver = m_array_;

         try
         {
            m_array_ = other.size() ? new T[other.size()]() : nullptr;
            std::copy(other.m_array_, other.m_array_ + other.size(), m_array_);
         }
         catch (std::exception& e)
         {
            if (m_array_saver != m_array_)
            {
               delete[] m_array_;
```

```
                m_array_ = m_array_saver;
            }

            throw;
        }

        delete[] m_array_saver;
        m_size_ = other.size();
    }

    return *this;
}

Array& operator=(Array&& other) noexcept
{
    delete[] m_array_;
    m_size_ = 0;

    std::swap(m_array_, other.m_array_);
    std::swap(m_size_, other.m_size_);

    return *this;
}

~Array()
{
    delete[] m_array_;
}

size_t size() const
{
    return m_size_;
}

T& operator [](const size_t index)
{
    assert(index < m_size_);

    return m_array_[index];
}

public:
    T* m_array_;
    size_t m_size_;
};
```