**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Математического обеспечения и применения ЭВМ**

отчет

**по лабораторной работе № 1**

**по дисциплине «ООП»**

**Тема: Конструирование объектов**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Ковынев М.В. |
| Преподаватель |  | Терентьев А.О. |

Санкт-Петербург

2018

# **Цель работы**

Реализация конструкторов копирования, перемещения, перегрузки оператора присваивания для класса Array, который должен давать строгую гарантию безопасности исключений.

# **Постановка задачи**

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 *Т* 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*).

**Ход работы**

1. Данные класса Array хранятся как динамический массив. Для управления освобождением памяти тип массива исходного шаблона кода был заменен массивом на *std::unique\_ptr.*

*std::unique\_ptr* – умный указатель, который:

* получает единоличное владение объектом через его указатель, и
* разрушает объект через его указатель, когда *unique\_ptr* выходит из области видимости.

*Unique\_ptr* не может быть скопирован или задан через операцию присвоения. Неконстантный *unique\_ptr* может передать владение управляемым объектом другому указателю *unique\_ptr. Const std::unique\_ptr*  не может быть передан, ограничивая время жизни управляемого объекта областью, в которой указатель был создан. Когда *unique\_ptr* уничтожается, он удаляет объект с помощью *Deleter.* Таким образом, в данной лабораторной работе деструктор в классе Array не понадобится.

Существует две версии *std::unique\_ptr:*

1. управляет временем жизни одного объекта, например, созданного с помощью оператора *new*
2. управляет временем жизни массива, с длиной, определенной во время выполнения, созданного с помощью *new[]*.

Типичные случаи применения *std::unique\_ptr* включают:

* обеспечение безопасности исключений для классов и функций, которые управляют объектами с динамическим временем жизни, гарантируя удаление в случае нормального завершения и завершения по исключению
* передача владения динамически созданным объектом в функции
* получение владения динамически созданным объектом из функций

1. В класс добавлена реализация конструктора копирования и оператора присваивания:

Array& operator=(Array obj)

{

std::swap(m\_size, obj.m\_size);

std::swap(m\_array, obj.m\_array);

print("operator=");

return\*this;

}

Array(const Array& obj)

: m\_size(obj.m\_size)

, m\_array(m\_size ? new T[m\_size]() : nullptr)

{

for (size\_t i = 0; i < m\_size; ++i)

m\_array[i] = obj.m\_array[i];

print("copy constr");

}

Используя такой конструктор копирования и затем не выбрасывающий исключения *std::swap,* можно быть уверенным в том, что, если где-то произойдет исключение, то промежуточного состояния с непроинициализированной памятью не возникнет.

1. В класс добавлена реализация конструктора перемещения:

Array(Array&& obj)

{

m\_array = move(obj.m\_array);

swap(m\_size, obj.m\_size);

obj.m\_size = 0;

print("move constr");

}

Двойной амперсанд – это ссылка на rvalue, который будет уничтожен после вызова. Таким образом мы перенесем все содержимое rvalue в область хранения данных какого-то объекта класса Array, после чего rvalue удалится.

Полный код с проведением различных тестов приведен в приложении A.

# **Заключение**

В ходе данной лабораторной работы рассмотрена семантика *commit-or-rollback*, реализованы конструкторы копирования и перемещения, также перегружен оператор присваивания в классе *Array*. Таким образом, разработанный класс *Array* дает гарантию безопасности исключений.

# **ПРИЛОЖЕНИЕ A**

**Код программы**

#include <assert.h>

#include <algorithm> // std::copy

#include <cstddef> // size\_t

#include <iostream>

#include <memory>

using namespace std;

template<typename T>

class Array

{

private:

size\_t m\_size;

std::unique\_ptr <T[]> m\_array;

void print(const char\* message)

{

cout << "[" << this << "] " << message << "\n";

}

public:

// (default) constructor

Array(const size\_t size = 0)

: m\_size(size)

, m\_array(m\_size ? new T[m\_size]() : nullptr)

{

print("default constr");

}

//Конструктор копирования

Array(const Array& obj)

: m\_size(obj.m\_size)

, m\_array(m\_size ? new T[m\_size]() : nullptr)

{

for (size\_t i = 0; i < m\_size; ++i)

m\_array[i] = obj.m\_array[i];

print("copy constr");

}

//Оператор присваивания

Array& operator=(Array obj)

{

std::swap(m\_size, obj.m\_size);

std::swap(m\_array, obj.m\_array);

print("operator=");

return\*this;

}

//Конструктор перемещения

Array(Array&& obj)

{

m\_array = move(obj.m\_array);

swap(m\_size, obj.m\_size);

obj.m\_size = 0;

print("move constr");

}

const size\_t size() const

{

return m\_size;

}

T& operator [](const size\_t index)

{

assert(index < m\_size);

return m\_array[index];

}

void print()

{

cout << "\nArray\n";

for (int i=0; i< m\_size; i++)

cout << m\_array[i] << " ";

cout << endl;

}

};

Array<int> GetRValue(Array<int> A) {

return A;

}

int main()

{

cout << "v1 initial: ";

Array <int> v1(10);

cout << "v2 initial: ";

Array <int> v2;

cout << "############################\n";

cout << "Assigment\n\n";

cout << "v1 before copy:";

v1.print();

cout << "v2 before copy:";

v2.print();

cout << "Start Copy\n";

v2 = v1;

cout << "End copy\n" << endl;

cout << "v1 after copy:";

v1.print();

cout << "v2 after copy:";

v2.print();

cout << "############################\n";

cout << "Copy\n\n";

cout << "v3 initial: ";

Array <int> v3(20);

cout << "Start Copy\n";

cout << "v4 initial: ";

Array <int> v4(v3);

cout << "End copy\n" << endl;

cout << "v3 after copy:";

v3.print();

cout << "v4 after copy:";

v4.print();

cout << "############################\n";

cout << "Move\n\n";

cout << "Start Move\n";

cout << "v5 initial: ";

Array <int> v5 = GetRValue(v4);

cout << "End Move\n" << endl;

cout << "v5 after move:";

v5.print();

cout << "v4 after move:";

v4.print();

cout << "############################\n";

cout << "Empty copy\n\n";

cout << "v6 initial: ";

Array <int> v6;

cout << "Start Copy\n";

cout << "v7 initial: ";

Array <int> v7 = v6;

cout << "End Copy\n" << endl;

system("pause");

return 0;

}