Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Объектно-ориентированное программирование»

Студент: С. М. Бокоч
Преподаватель:
Группа: М8О-204Б

Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №3

Задача: Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий все три фигуры, согласно варианту задания. Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Класс-контейнер должен содержать объекты, используя std::shared ptr<...>.
- Класс-контейнер должен иметь метод по добалению фигуры в контейнер.
- Класс-контейнер должен иметь методы по получению фигуры из контейнера.
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера.
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток ostream.
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположены в раздельных файлах: отдельно заголовки(.h), отдельно описание методов (.cpp).

Фигуры. Квадрат, трапеция, прямоугольник.

Контейнер. Массив.

1 Теория

Умный указатель – класс (обычно шаблонный), имитирующий интерфейс обычного указателя и добавляющий некую новую функциональность, например, проверку границ при доступе или очистку памяти. В данной лабораторной работе я использовал **shared_ptr** из библиотеки STL. **shared_ptr** – умный указатель с подсчитанными ссылками. Используется, когда необходимо присвоить один необработанный указатель нескольким владельцам,например, когда копия указателя возвращается из контейнера, но требуется сохранить оригинал. Необработанный указатель не будет удален до тех пор, пока все владельцы shared_ptr не выйдут из области или не откажутся от владения

2 Листинг

```
1  //TArray.cpp
2  #include <memory>
3  #include "TArray.h"
4  TArray::TArray() {
5      _data = new std::shared_ptr<Figure>[DEFAULT_CAPACITY];
6      _capacity = DEFAULT_CAPACITY;
7      _size = 0;
8  }
9  TArray::TArray(const size_t &sizeArr) {
10     _data = new std::shared_ptr<Figure>[sizeArr];
11     for (int i = 0; i < sizeArr; i++) {
12         _data[i] = nullptr;
13     }
14     _capacity = sizeArr;
15     _size = 0;
16 }
17 TArray::TArray(TArray& orig) {
18     _data = new std::shared_ptr<Figure>[orig._capacity];
19     this->_size = orig._size;
20     this->_capacity = orig._capacity;
21     for (size_t index = 0; index < _size; index++) {
22         _data[index] = orig._data[index];
23     }
24 }
25 bool TArray::Empty() {
26     return _size == 0;
27 }
28 std::ostream &operator<<(std::ostream &os, const TArray &objArr) {
29     for (size_t index = 0; index < objArr._size; ++index) {
30         if (&objArr[index] != nullptr)
31             os << index << "\t";
32         objArr[index]->Print();
33     }
34     return os;
35 }
36 void TArray::Push_back(std::shared_ptr<Figure> &temp) {
37     if (_size == _capacity) {
38         _capacity *= 2;
39         std::shared_ptr<Figure> *copyArr = new std::shared_ptr<Figure>[_capacity];
40         for (size_t index = 0; index < _size; ++index) {
41             copyArr[index] = this->_data[index];
42         }
43         delete [] _data;
44         _data = copyArr;
45     }
46     this->_data[_size++] = temp;
47 }
```

```
48 bool TArray::Delete(const size_t index) {
49     std::shared_ptr<Figure> *tCopy = new std::shared_ptr<Figure>[_capacity];
50     int j = 0;
51     bool flag = false;
52     for (int i = 0; i < _size; i++) {
53         if (i!=index) {
54             tCopy[j++] = _data[i];
55         }
56         else {
57             flag = true;
58         }
59     }
60     _size--;
61     delete [] _data;
62     _data = tCopy;
63     return flag;
64 }
65 std::shared_ptr<Figure>& TArray::operator[](size_t index) const{
66     if (index < _size)
67         return _data[index];
68     return _data[0];
69 }
70 std::shared_ptr<Figure>& TArray::operator[](size_t index) {
71     if (index < _size)
72         return _data[index];
73     return _data[0];
74 }
75 size_t TArray::Size() const{
76     return this->_size;
77 }
78 size_t TArray::Capacity() const {
79     return this->_capacity;
80 }
81 TArray::~TArray() {
82     delete[] _data;
83 }
```

3 Выводы

Умные указатели, безусловно, являются необходимыми инструментами, если приходится иметь дело с динамическим выделением памяти. Хоть я и понимаю как они работают и для себя делал несколько реализаций, все равно почему с ними программа работает без ошибок – остается загадкой. Также важно знать чем указатели отличаются друг от друга, потому что бывают моменты, когда обойтись одним лишь shared_ptr нельзя.