

**Московский авиационный институт  
(национальный исследовательский университет)**

**Факультет информационных технологий и прикладной  
математики**

**Кафедра вычислительной математики и программирования**

**Лабораторная работа №6 по курсу «Объектно-ориентированное  
программирование»**

Студент: Болотин А.Н.  
Преподаватель: Поповкин А. В.  
Группа: 08-207  
Вариант: 19  
Дата:  
Оценка:  
Подпись:

**Москва, 2017**

# Лабораторная работа №6

## 1 Цель работы

- Закрепление навыков по работе с памятью в C++.
- Создание аллокаторов памяти для динамических структур данных.

## 2 Задача

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР №5) спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Аллокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-ого уровня, согласно варианту задания).

Для вызова аллокатора должны быть переопределены операторы new и delete у классов-фигур.

**Фигуры:** прямоугольник, ромб, трапеция.

**Контейнер 1:** N-дерево. **Контейнер 2:** Массив.

## 3 Описание

Аллокатор памяти – часть программы (как прикладной, так и операционной системы), обрабатывающая запросы на выделение и освобождение оперативной памяти или запросы на включение заданной области памяти в адресное пространство процессора.

Основное назначение аллокатора памяти в первом смысле – реализация динамической памяти. В языке С динамическое выделение памяти производится через функцию `malloc`.

Программисты должны учитывать последствия динамического выделения памяти и дважды обдумать использование функции `malloc` или оператора `new`. Легко убедить себя, что вы не делаете так уж много аллокаций, а значит большого значения это не имеет, но такой тип мышления распространяется лавиной по всей команде и приводит к медленной смерти. Фрагментация и потери в производительности, связанные с использованием динамической памяти, не будучи пресеченными в зародыше, могут иметь катастрофические трудноразрешаемые последствия в вашем дальнейшем цикле разработки. Проекты, где управление и распределение памяти не продумано надлежащим образом, часто страдают от случайных сбоев после длительной сессии из-за нехватки памяти и стоят сотни часов работы программистов, пытающихся освободить память и реорганизовать ее выделение.

## 4 Исходный код

Описание классов фигур и класса-контейнера остается неизменным.

```
1 class TAllocationBlock
2 {
3 public:
4     TAllocationBlock(size_t size, size_t count);
5     void *allocate();
6     void deallocate(void *pointer);
7     bool hasFreeBlocks();
8
9     ~TAllocationBlock();
10
11 private:
12     size_t size;
13     size_t count;
14
15     char * usedBlocks;
16     void ** freeBlocks;
17
18     size_t freeCount;
```

19 || };

## 5 Выводы

В результате выполнения был разработан аллокатор памяти для дерева, оптимизирован вызов операции malloc, переопределены операторы new-delete. Для хранения свободных блоков использовался массив памяти.