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: С. М. Бокоч
Преподаватель:
Группа: М8О-204Б
Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №6

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. Квадрат, трапеция, прямоугольник.

Контейнер первого уровня. Массив. **Контейнер первого уровня.** Список.

1 Теория

Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. std::allocator – пример реализации аллокатора из стандартной библиотеки, просто использует new и delete, которые обычно обращаются к системным вызовам malloc и free. Программист обладает преимуществом над стандартным аллокатором, он знает какое количество памяти будет выделятся чаще, как она будет связана. Хорошим способом оптимизации программы будет уменьшение количества системных вызовов, которые происходят при аллокации. Аллоцировав сразу большой отрезок памяти и распределяя его, можно добиться положительных эффектов.

2 Листинг

```
1  //TAllocationBlock.h
2  #ifndef TALLOCATIONBLOCK_H
3  #define TALLOCATIONBLOCK_H
4
5  #include <iostream>
6  #include <cstdlib>
7  #include "TList.h"
8
9  typedef unsigned char Byte;
10 typedef void * VoidPtr;
11
12 template<class T>
13 class TList;
14
15 class TAllocationBlock
16 {
17 public:
18     TAllocationBlock(size_t size, size_t count);
19     void *Allocate();
20     void Deallocate(void *ptr);
21     bool Empty();
22     size_t Size();
23
24     virtual ~TAllocationBlock();
25
26 private:
27     Byte *_used_blocks;
28     TList<VoidPtr> _free_blocks;
29 };
30
31 #endif /* TALLOCATIONBLOCK_H * */
32 //TAllocationBlock.cpp
33 #include "TAllocationBlock.h"
34 #include <iostream>
35 TAllocationBlock::TAllocationBlock(size_t size, size_t count)
36 {
37     _used_blocks = (Byte *)malloc(size * count);
38     void * ptr;
39     for (size_t i = 0; i < count; ++i) {
40         ptr = _used_blocks + i * size;
41         _free_blocks.Push(ptr);
42     }
43 }
44 void *TAllocationBlock::Allocate()
45 {
46     if (!_free_blocks.IsEmpty()) {
47         void * res = _free_blocks.Top();
```

```
48          _free_blocks.Pop();
49          return res;
50      }
51      else {
52          throw std::bad_alloc();
53      }
54  }
55  void TAllocationBlock::Deallocate(void *ptr)
56  { _free_blocks.Push(ptr);}
57  bool TAllocationBlock::Empty()
58  {return _free_blocks.IsEmpty();}
59  size_t TAllocationBlock::Size()
60  { return _free_blocks.GetLength(); }
61  TAllocationBlock::~TAllocationBlock()
62  {
63      free(_used_blocks);
64      std::cout << "allocator finished it\'s work\n";
65  }
66  //TList.cpp
67  #include <iostream>
68  #include <memory>
69  #include "TList.h"
70  template <class T>
71  TList<T>::TList()
72  {
73      head = nullptr;
74      count = 0;
75  }
76  template <class T>
77  void TList<T>::Push(const T &item)
78  {
79      TListItem<T> *tmp = new TListItem<T>(item, head);
80      head = tmp;
81      ++count;
82  }
83  template <class T>
84  bool TList<T>::IsEmpty() const
85  { return !count;}
86  template <class T>
87  size_t TList<T>::GetLength() const
88  {
89      return count;
90  }
91  template <class T>
92  void TList<T>::Pop()
93  {
94      if (head) {
95          TListItem<T> *tmp = &head->GetNext();
96          delete head;
```

```
97          head = tmp;
98          --count;
99      }
100 }
101 template <class T>
102 T &TList<T>::Top()
103 { return head->Pop();}
104 template <class T>
105 TList<T>::~TList()
106 {
107     for (TListItem<T> *tmp = head, *tmp2; tmp; tmp = tmp2) {
108         tmp2 = &tmp->GetNext();
109         delete tmp;
110     }
111 }
112 typedef unsigned char TByte;
113 template class
114 TList<void *>;
```

3 Выводы

в данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки создания аллокаторов. Добавил аллокатор и упрощенный список, в котором будут хранится адреса использованных/свободных блоков. Это очень важный опыт и в дальнейшем он пригодится мне для написания курсового проекта по ОС.