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: М. В. Спиридонов
Преподаватель:
Группа: М8О-206Б

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, треугольник, прямоугольник.

Контейнер первого уровня. Массив.

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  //main.cpp
2  #include <iostream>
3  #include "TVector.h"
4  #include "Rectangle.h"
5  #include "Triangle.h"
6  #include "Square.h"
7  #include <chrono>
8
9  void Menu () {
10     std::cout << "Choose an operation:" << std::endl;
11     std::cout << "1) Add triangle" << std::endl;
12     std::cout << "2) Add rectangle" << std::endl;
13     std::cout << "3) Add square" << std::endl;
14     std::cout << "4) Get by Index" << std::endl;
15     std::cout << "5) Print vector" << std::endl;
16     std::cout << "0) Exit" << std::endl;
17 }
18
19 int main () {
20
21     TVector<int> intV;
22     clock_t start = clock();
23     int *j = nullptr;
24     for (int i = 0; i < 100000; ++i) {
25         j=new int(10);
26         intV.FastPushBack(*j);
27     }
28     clock_t end = clock();
29     double time = (double)(end-start);
30     time/=CLOCKS_PER_SEC;
31
32     std::cout << time;
33     start = clock();
34     TVector<TVectorItem<int>> *intB = new TVector<TVectorItem<int>>();
35     for (int i = 0; i < 100000; ++i) {
36         TVectorItem<int> *b = new TVectorItem<int>(*j);
37         intB->FastPushBack(*b);
38     }
39     end = clock();
40     time = (double)(end-start)/CLOCKS_PER_SEC;
41     std::cout << time;
42     //delete intV;
43     /*
44     do {
45         Menu();
46         std::cin >> action;
47         switch (action) {
```

```
            case 1:
                delete ptr;
                delete tr;
                ptr = new Triangle(std::cin);
                tr = new TVectorItem <Figure>(ptr);
                tVector.FastPushBack(*tr);
                break;
            case 2:
                delete ptr;
                delete tr;
                ptr = new Rectangle(std::cin);
                tr = new TVectorItem <Figure>(ptr);
                tVector.FastPushBack(*tr);
                break;
            case 3:
                delete ptr;
                delete tr;
                ptr = new TSquare(std::cin);
                tr = new TVectorItem <Figure>(ptr);
                tVector.FastPushBack(*tr);
                break;
            case 4:
                std::cin >> action;
                if (action <= tVector.Capacity() - 1)
                    std::cout << tVector[action] << std::endl;
                else {
                    std::cout << "Index out of range";
                }
                break;
            case 5:
                for (auto i: tVector) {
                    std::cout << i;
                }
                break;
            case 0:
                break;
            default:
                std::cout << "Incorrect command" << std::endl;;
                break;
        }
    } while (action);
    delete ptr;
    delete tr;*/
    return 0;
}
//TAllocationBlock.h
#ifndef TALLOCATIONBLOCK_H
#define TALLOCATIONBLOCK_H

```

```
97  #include <cstdlib>
98
99  class TAllocationBlock {
100 public:
101     TAllocationBlock (size_t size, size_t count);
102
103     void *allocate ();
104
105     void *allocateSome(size_t count);
106
107     void deallocate (void *pointer);
108
109     void deallocateSome (void *pointer, size_t count);
110
111     bool has_free_blocks ();
112
113     virtual ~TAllocationBlock ();
114
115 private:
116     size_t _size;
117     size_t _count;
118
119     char *_used_blocks;
120     void **_free_blocks;
121
122     size_t _free_count;
123 };
124
125
126 #endif //PROG_TALLOCATIONBLOCK_H
127
128 //TAllocationBlock.cpp
129
130 #include "TAllocationBlock.h"
131 #include <iostream>
132
133 TAllocationBlock::TAllocationBlock (size_t size, size_t count) : _size(size), _count(
        count) {
134     _used_blocks = (char *) malloc(_size * _count);
135     _free_blocks = (void **) malloc(sizeof(void *) * _count);
136
137     for (size_t i = 0; i < _count; i++) {
138         _free_blocks[i] = _used_blocks + i * _size;
139     }
140     _free_count = _count;
141     std::cout << "TAllocationBlock: Memory init" << std::endl;
142 }
143
144 void *TAllocationBlock::allocate () {
```

```
145     void *result = nullptr;
146 
147     if (has_free_blocks()) {
148         result = _free_blocks[_free_count - 1];
149         _free_count--;
150         //std::cout << "TAllocationBlock: Allocate " << (_count - _free_count) << " of
                " << _count << std::endl;
151     } else {
152         //std::cerr << "TAllocationBlock: No memory exception :-)" << std::endl;
153     }
154 
155     return result;
156 }
157 
158 void *TAllocationBlock::allocateSome (size_t count) {
159     void *result = nullptr;
160     void *first = nullptr;
161     for (int i = 0; i < count; ++i) {
162         result = allocate();
163 
164         if (result != nullptr && i == 0) {
165             first = result;
166         }
167     }
168     if(has_free_blocks()) {
169         std::cout << "TAllocationBlock: Allocate " << (_count - _free_count) << " of "
                << _count << std::endl;
170     }else{
171         std::cerr << "TAllocationBlock: No memory exception :-)" << std::endl;
172     }
173     return first;
174 }
175 
176 void TAllocationBlock::deallocate (void *pointer) {
177 
178     if(_free_count<_count){
179         _free_blocks[_free_count] = pointer;
180         _free_count++;
181 
182     }
183     else{
184         std::cerr << "TAllocationBlock: Failed Deallocate block " << std::endl;
185     }
186 
187 }
188 
189 bool TAllocationBlock::has_free_blocks () {
190     return _free_count > 0;
191 }
```

```
192
193 TAllocationBlock::~TAllocationBlock () {
194
195     if (_free_count < _count) std::cout << "TAllocationBlock: Memory leak?" << std::
        endl;
196     else std::cout << "TAllocationBlock: Memory freed" << std::endl;
197     delete _free_blocks;
198     delete _used_blocks;
199 }
200
201 void TAllocationBlock::deallocateSome (void *pointer, size_t count) {
202     for (int i = 0; i < count; ++i) {
203         deallocate(pointer);
204     }
205     std::cout << "TAllocationBlock: Deallocate blocks ["<<count<<"] " << std::endl;
206 }
207 //TVectorItem.h
208 #ifndef TVECTORITEM_H
209 #define TVECTORITEM_H
210
211 #include <memory>
212 #include "iostream"
213 #include "TAllocationBlock.h"
214
215 template <class T>
216 class TVectorItem {
217 public:
218
219     //TVectorItem();
220
221     TVectorItem ();
222
223     TVectorItem (T &item);
224
225     TVectorItem (T *item);
226
227     template <class A>
228     friend std::ostream &operator << (std::ostream &os, const TVectorItem <A> &obj);
229
230     void *operator new (size_t size);
231
232     void *operator new[] (size_t count);
233
234     void operator delete (void *p);
235
236     void operator delete[] (void *p, size_t count);
237
238     virtual ~TVectorItem ();
239
```

```
240 private:
241
242     static TAllocationBlock _stackItemAllocator;
243
244     T* _item;
245 };
246
247 #include "TVectorItem.hpp"
248
249 #endif //PROG_TVECTORITEM_H
250 //TVectorItem.hpp
251 #ifndef TVECTORITEM_HPP
252 #define TVECTORITEM_HPP
253
254 #include "TVectorItem.h"
255
256 template <class T>
257 TVectorItem <T>::TVectorItem () {
258     this->_item = nullptr;
259     //std::cout << "TVectorItem: null created" << std::endl;
260 }
261
262 template <class T>
263 TVectorItem <T>::TVectorItem (T &item) {
264     this->_item = &item;
265     //std::cout << "TVectorItem: created" << std::endl;
266 }
267
268 template <class T>
269 TVectorItem <T>::TVectorItem (T *item) {
270     this->_item = item;
271     //std::cout << "TVectorItem: created" << std::endl;
272 }
273
274 template <class T>
275 TAllocationBlock TVectorItem <T>::_stackItemAllocator (sizeof(T),1600000);
276
277 template <class A>
278 std::ostream &operator << (std::ostream &os, const TVectorItem <A> &obj) {
279     //os << "[" << *obj._item << "]" << std::endl;
280     obj._item->Print();
281     return os;
282 }
283
284 template <class T>
285 void *TVectorItem <T>::operator new (size_t size) {
286     return _stackItemAllocator.allocate();
287 }
288
```

```
289 template <class T>
290 void *TVectorItem <T>::operator new [](size_t count) {
291     return _stackItemAllocator.allocateSome(count);
292 }
293 
294 template <class T>
295 void TVectorItem <T>::operator delete (void *p) {
296     _stackItemAllocator.deallocate(p);
297 }
298 
299 template <class T>
300 void TVectorItem <T>::operator delete[] (void *p, size_t count) {
301     _stackItemAllocator.deallocateSome(p,count);
302 }
303 
304 template <class T>
305 TVectorItem <T>::~TVectorItem () {
306     //std::cout << "TVectorItem deleted" << std::endl;
307 }
308 
309 #endif //PROG_TVECTORITEM_HPP
```

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.