Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №7 по курсу «Объектно-ориентированное программирование»

Студент: М. В. Спиридонов
Преподаватель:
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №7

Задача:
Целью лабораторной работы является:

1. Создание сложных динамических структур данных.
2. Закрепление принципа ООП.

"Хранилище объектов"представляет собой контейнер(массив), в котором каждый элемент контейнера является динамическая структура(список). Таким образом, у нас получается контейнер в контейнере. Элементов второго контейнера является объект-фигура, определенная вариантом задания. При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше 5. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта. При удалении объектов должно выполянться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалиться.

Фигуры. Квадрат, треугольник, прямоугольник.

Контейнер первого уровня. Массив.

Контейнер второго уровня. Массив.

1 Теория

Принцип открытости/закрытости — принцип объектно-ориентированного программирования, устанавливающий следующее положение: «программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения»; это означает, что такие сущности могут позволять менять свое поведение без изменения их исходного кода. Это особенно значимо в производственной среде, когда изменения в исходном коде потребуют проведение пересмотра кода, модульного тестирования и других подобных процедур, чтобы получить право на использования его в программном продукте. Код, подчиняющийся данному принципу, не изменяется при расширении и поэтому не требует таких трудозатрат.

Это просто означает, что класс должен быть легко расширяемый без изменения самого класса.

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
8  void Menu () {
9      std::cout << "Choose an operation:" << std::endl;
10     std::cout << "1) Add triangle" << std::endl;
11     std::cout << "2) Add rectangle" << std::endl;
12     std::cout << "3) Add square" << std::endl;
13     std::cout << "4) Get by Index" << std::endl;
14     std::cout << "5) Print vector" << std::endl;
15     std::cout << "0) Exit" << std::endl;
16 }
17
18 int main () {
19     int constT = 10;
20     TVector <TVector <std::shared_ptr <Figure>>> toLevelVector(constT);
21     int toLevelIndex = 0;
22     TVector <std::shared_ptr <Figure>> tVector(constT);
23     for (int j = 0; j < constT; ++j) {
24         toLevelVector.FastPushBack(tVector);
25     }
26     int action = 0;
27     auto tmp = std::shared_ptr <Figure>(new Triangle(3, 4, 5));
28
29     do {
30         Menu();
31         std::cin >> action;
32         switch (action) {
33             case 1:
34                 tmp = std::shared_ptr <Figure>(new Triangle(std::cin));
35                 if (toLevelVector[toLevelIndex].Capacity() >= 5) {
36                     ++toLevelIndex;
37                 }
38                 toLevelVector[toLevelIndex].FastPushBack(tmp);
39
40                 break;
41             case 2:
42                 tmp = std::shared_ptr <Figure>(new Rectangle(std::cin));
43                 if (toLevelVector[toLevelIndex].Capacity() >= 5) {
44                     ++toLevelIndex;
45                 }
46                 toLevelVector[toLevelIndex].FastPushBack(tmp);
47                 break;
```

```
48              case 3:
49                  tmp = std::shared_ptr <Figure>(new TSquare(std::cin));
50                  if (toLevelVector[toLevelIndex].Capacity() >= 5) {
51                      ++toLevelIndex;
52                  }
53                  toLevelVector[toLevelIndex].FastPushBack(tmp);
54                  break;
55              case 4:
56                  std::cin >> action;
57                  int index;
58                  std::cin >> index;
59                  if ((action <= toLevelVector.Capacity() - 1)
60                      && (index <= toLevelVector[action].Capacity() - 1))
61
62                      std::cout << toLevelVector[action][index] << std::endl;
63                  else {
64                      std::cout << "Index out of range";
65                  }
66                  break;
67              case 5:
68                  for (auto &i: toLevelVector) {
69                      for(auto &k: i){
70                          k->Print();
71                      }
72                  }
73                  break;
74              case 6:
75                  tVector.Sort(0, tVector.Capacity() - 1);
76                  break;
77              case 0:
78                  break;
79              default:
80                  std::cout << "Incorrect command" << std::endl;;
81                  break;
82          }
83      } while (action);
84
85
86      tVector.Sort(0,6);
87      std::cout << std::endl;
88      for (const auto &i: tVector) {
89          i->Print();
90      }
91      return 0;
92  }
93  //TVector.h
94  #ifndef TVECTOR_H
95  #define TVECTOR_H
96
```

```
97  #include "iostream"
98  #include "TAllocationBlock.h"
99  #include "TVectorItem.h"
100 //#include <memory>
101
102 template <class T>
103 class Iter {
104 public:
105     Iter (int n, T *arr) {
106         _n = n;
107         _arr = arr;
108     }
109
110     T &operator * () {
111         return _arr[_n];
112     }
113
114     T &operator -> () {
115         return _arr[_n];
116     }
117
118     void operator ++ () {
119         _n++;
120     }
121
122     Iter operator ++ (int) {
123         ++(*this);
124         return &this;
125     }
126
127     bool operator == (Iter const &i) {
128         return _n == i._n &&
129                _arr == i._arr;
130     }
131
132     bool operator != (Iter const &i) {
133         return !(*this == i);
134     }
135
136 private:
137     int _n;
138     T *_arr;
139 };
140
141 template <class T>
142 class TVector {
143 public:
144
145     void FastPushBack (T &data);
```

```
146
147     void Clear ();
148
149     T &operator [] (int index);
150
151     int Size () const;
152
153     int Capacity () const;
154
155     Iter <T> begin ();
156
157     Iter <T> end ();
158
159     TVector &operator = (const TVector <T> &inp);
160
161     TVector ();
162
163     void Sort(int low, int hight);
164
165     TVector (const TVector <T> &inp);
166
167     friend std::ostream &operator << (std::ostream &os, const TVector <T> &tVector);
168
169     explicit TVector (int n);
170
171     ~TVector ();
172
173 private:
174     void swap(T* a, T* b);
175     int partition (int low, int high);
176     //static TAllocationBlock vector_allocator;
177     T *privateArray;
178     int privateArraySize;
179     int privateArrayOccupiedSize;
180     static int const SIZE_DIFFERENCE = 1;
181     static int const DEFAULT_INT_VALUE = 0;
182 };
183
184 #include "TVector.hpp"
185
186 #endif //TVECTOR_H
187 //TVector.hpp
188 #ifndef TVECTOR_HPP
189 #define TVECTOR_HPP
190
191 #include "TVector.h"
192
193 template <class T>
194 std::ostream &operator << (std::ostream &os, const TVector <T> &tVector) {
```

```
195     for (int i = 0; i < tVector.Capacity(); ++i) {
196         os << tVector[i]->Print() << std::endl;
197     }
198     return os;
199 }
200 template <class T>
201 void TVector<T>::swap(T* a, T* b)
202 {
203     T t = *a;
204     *a = *b;
205     *b = t;
206 }
207 
208 template <class T>
209 int TVector<T>::partition (int low, int high)
210 {
211     double pivot = privateArray[high]->Square(); // pivot
212     int i = (low - 1); // Index of smaller element
213 
214     for (int j = low; j <= high- 1; j++)
215     {
216         // If current element is smaller than or
217         // equal to pivot
218         if (privateArray[j]->Square() <= pivot)
219         {
220             i++; // increment index of smaller element
221             swap(&privateArray[i], &privateArray[j]);
222         }
223     }
224     swap(&privateArray[i + 1], &privateArray[high]);
225     return (i + 1);
226 }
227 
228 template <class T>
229 void TVector<T>::Sort (int low, int high) {
230 
231     //clock_t start = clock();
232     if (low <= high)
233     {
234         /* pi is partitioning index, arr[p] is now
235            at right place */
236         int pi = partition(low, high);
237 
238         // Separately sort elements before
239         // partition and after partition
240         Sort(low, pi - 1);
241         Sort(pi + 1, high);
242     }
243     //clock_t end = clock();
```

```
244     //printf("\nCounter SortTime = %lf\n", (double)(end - start) / CLOCKS_PER_SEC);
245 }
246
247 template <class T>
248 void TVector <T>::FastPushBack (T &data) {
249     if (privateArraySize == privateArrayOccupiedSize) {
250         privateArraySize *= 2;
251         T *result = new T[privateArraySize];
252
253         for (int index = DEFAULT_INT_VALUE; index <= privateArrayOccupiedSize; index++)
                {
254             if (index != privateArrayOccupiedSize) {
255                 result[index] = privateArray[index];
256             } else {
257                 result[index] = data;
258                 break;
259             }
260         }
261         delete[] privateArray;
262         privateArray = result;
263         privateArrayOccupiedSize++;
264     } else {
265         privateArray[privateArrayOccupiedSize] = data;
266         privateArrayOccupiedSize++;
267     }
268 }
269
270 template <class T>
271 void TVector <T>::Clear () {
272     delete[] privateArray;
273     privateArraySize = DEFAULT_INT_VALUE;
274     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
275     privateArray = new T[privateArraySize];
276 }
277
278 template <class T>
279 T &TVector <T>::operator [] (int index) {
280     return this->privateArray[index];
281 }
282
283 template <class T>
284 int TVector <T>::Size () const {
285     return privateArraySize;
286 }
287
288 template <class T>
289 int TVector <T>::Capacity () const {
290     return this->privateArrayOccupiedSize;
291 }
```

```
292
293 template <class T>
294 TVector <T> &TVector <T>::operator = (const TVector <T> &inp) {
295     return *this;
296 }
297
298 template <class T>
299 TVector <T>::TVector () {
300     privateArraySize = DEFAULT_INT_VALUE;
301     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
302     privateArray = new T[privateArraySize];
303 }
304
305 template <class T>
306 TVector <T>::TVector (const int n) {
307     privateArraySize = n;
308     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
309     privateArray = new T[privateArraySize];
310 }
311
312 template <class T>
313 TVector <T>::TVector (const TVector <T> &inp) {
314     privateArraySize = inp.privateArraySize;
315     privateArrayOccupiedSize = inp.privateArrayOccupiedSize;
316     privateArray = inp.privateArray;
317 }
318
319 template <class T>
320 Iter<T> TVector<T>::begin () {
321     return Iter<T>(0,privateArray);
322 }
323
324 template <class T>
325 Iter<T> TVector<T>::end () {
326     return Iter<T>(privateArrayOccupiedSize,privateArray);
327 }
328
329 template <class T>
330 TVector <T>::~TVector () {
331     delete[] privateArray;
332     privateArraySize = DEFAULT_INT_VALUE;
333     privateArrayOccupiedSize = DEFAULT_INT_VALUE;
334 }
335
336 #endif // TVECTOR_HPP
```

3 Выводы

В данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки создания сложных динамических структур. Применил на практике принцип OCP. Создал сложное хранилище данных, автосортируемый по площади, с возможностью удаления по критериям. Это было очень трудно, особенно, сортировать в элементы в списке.