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

Москва, 2017

Лабораторная работа №7

Задача:
Целью лабораторной работы является:

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

"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, трапеция, прямоугольник.

Контейнер первого уровня. Массив.

Контейнер первого уровня. Список.

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

```
//TList.h
#ifndef TLIST_H
#define TLIST_H
#include <memory>
#include <iostream>
#include "TAllocator.h"
#include <future>
#include <thread>
#include <functional>

template <typename T> class TList
{
private:
    class TNode {
    public:
        TNode();
        TNode(const std::shared_ptr<T>&);
        auto GetNext() const;
        auto GetItem() const;
        std::shared_ptr<T> item;
        std::shared_ptr<TNode> next;

        void* operator new(size_t);
        void operator delete(void*);
        static TAllocator nodeAllocator;
    };

    template <typename N, typename M>
    class TIterator {
    private:
        N nodePtr;
    public:
        TIterator(const N&);
        std::shared_ptr<M> operator* ();
        std::shared_ptr<M> operator-> ();
        void operator ++ ();

        bool operator == (const TIterator&);
        bool operator != (const TIterator&);
    };

    int length;

    std::shared_ptr<TNode> head;
    auto psort(std::shared_ptr<TNode>&);
    auto pparsort(std::shared_ptr<TNode>& head);
    auto partition(std::shared_ptr<TNode>&);
```

```
48
49  public:
50      TList();
51      bool PushFront(const std::shared_ptr<T>&);
52      bool Push(const std::shared_ptr<T>&, const int);
53      bool PopFront();
54      bool Pop(const int);
55      bool IsEmpty() const;
56      int GetLength() const;
57      auto& getHead();
58      auto&& getTail();
59      void sort();
60      void parSort();
61
62      TIterator<std::shared_ptr<TNode>, T> begin() {return TIterator<std::shared_ptr<
            TNode>, T>(head->next);};
63      TIterator<std::shared_ptr<TNode>, T> end() {return TIterator<std::shared_ptr<TNode
            >, T>(nullptr);};
64
65      template <typename A> friend std::ostream& operator<< (std::ostream&, TList<A>&);
66  };
67
68  #include "TList.hpp"
69  #include "TIterator.hpp"
70  #endif
71  //TList.cpp
72  #ifdef TLIST_H
73  template <typename T> TList<T>::TNode::TNode()
74  {
75      item = std::shared_ptr<T>();
76      next = nullptr;
77  }
78
79  template <typename T> TList<T>::TNode::TNode(const std::shared_ptr<T>& obj)
80  {
81      item = obj;
82      next = nullptr;
83  }
84
85  template <typename T> TAllocator TList<T>::TNode::nodeAllocator(sizeof(TList<T>::TNode
        ), 100);
86
87  template <typename T> void* TList<T>::TNode::operator new(size_t size)
88  {
89      return nodeAllocator.allocate();
90  }
91
92  template <typename T> void TList<T>::TNode::operator delete(void* ptr)
93  {
```

```
94      nodeAllocator.deallocate(ptr);
95  }
96
97  template <typename T> TList<T>::TList()
98  {
99      head = std::make_shared<TNode>();
100     length = 0;
101 }
102
103 template <typename T> bool TList<T>::IsEmpty() const
104 {
105     return this->length == 0;
106 }
107
108 template <typename T> auto& TList<T>::getHead()
109 {
110     return this->head->next;
111 }
112
113 template <typename T> auto&& TList<T>::getTail()
114 {
115     auto tail = head->next;
116     while (tail->next != nullptr) {
117         tail = tail->next;
118     }
119
120     return tail;
121 }
122
123 template <typename T> int TList<T>::GetLength() const
124 {
125     return this->length;
126 }
127
128 template <typename T> bool TList<T>::PushFront(const std::shared_ptr<T>& obj)
129 {
130     auto Nitem = std::make_shared<TNode>(obj);
131     std::swap(Nitem->next, head->next);
132     std::swap(head->next, Nitem);
133     length++;
134
135     return true;
136 }
137
138 template <typename T> bool TList<T>::Push(const std::shared_ptr<T>& obj, int pos)
139 {
140     if (pos == 1 || length == 0)
141         return PushFront(obj);
142     if (pos < 0 || pos > length + 1)
```

```
143         return false;
144 
145     auto iter = head->next;
146     int i = 0;
147 
148     while (i < pos - 2) {
149         iter = iter->next;
150         i++;
151     }
152 
153     auto Nitem = std::make_shared<TNode>(obj);
154     std::swap(Nitem->next, iter->next);
155     std::swap(iter->next, Nitem);
156     length++;
157 
158     return true;
159 }
160 
161 template <typename T> bool TList<T>::PopFront()
162 {
163     if (IsEmpty())
164         return false;
165 
166     head->next = std::move(head->next->next);
167 
168     length--;
169 
170     return true;
171 }
172 
173 template <typename T> bool TList<T>::Pop(int pos)
174 {
175     if (pos < 1 || pos > length || IsEmpty())
176         return false;
177     if (pos == 1)
178         return PopFront();
179 
180     auto iter = head->next;
181     int i = 0;
182 
183     while (i < pos - 2) {
184         iter = iter->next;
185         i++;
186     }
187 
188     iter->next = std::move(iter->next->next);
189     length--;
190 
191     return true;
```

```
192 }
193
194 template <typename T> auto TList<T>::TNode::GetNext() const
195 {
196     return this->next;
197 }
198
199 template <typename T> auto TList<T>::TNode::GetItem() const
200 {
201     return this->item;
202 }
203
204 template <typename A> std::ostream& operator<< (std::ostream& os, const TList<A>& list
        )
205 {
206     if (list.IsEmpty()) {
207         os << "The list is empty!" << std::endl;
208         return os;
209     }
210
211     auto tmp = list.head->GetNext();
212     while(tmp != nullptr) {
213         tmp->GetItem()->Print();
214         tmp = tmp->GetNext();
215     }
216
217     return os;
218 }
219
220 template <typename T> auto TList<T>::psort(std::shared_ptr<TNode>& head)
221 {
222     if (head == nullptr || head->next == nullptr) {
223         return head;
224     }
225
226     auto partitionedEl = partition(head);
227     auto leftPartition = partitionedEl->next;
228     auto rightPartition = head;
229
230     partitionedEl->next = nullptr;
231
232     if (leftPartition == nullptr) {
233         leftPartition = head;
234         rightPartition = head->next;
235         head->next = nullptr;
236     }
237
238     rightPartition = psort(rightPartition);
239     leftPartition = psort(leftPartition);
```

```
240     auto iter = leftPartition;
241     while (iter->next != nullptr) {
242         iter = iter->next;
243     }
244 
245     iter->next = rightPartition;
246 
247     return leftPartition;
248 }
249 
250 template <typename T> auto TList<T>::partition(std::shared_ptr<TNode>& head)
251 {
252     if (head->next->next == nullptr) {
253         if (head->next->GetItem()->getSquare() > head->GetItem()->getSquare()) {
254             return head->next;
255         } else {
256             return head;
257         }
258     } else {
259         auto i = head->next;
260         auto pivot = head;
261         auto lastElSwapped = (pivot->next->GetItem()->getSquare()
262                         >= pivot->GetItem()->getSquare()) ? pivot->next : pivot;
263 
264         while ((i != nullptr) && (i->next != nullptr)) {
265             if (i->next->GetItem()->getSquare() >= pivot->GetItem()->getSquare()) {
266                 if (i->next == lastElSwapped->next) {
267                     lastElSwapped = lastElSwapped->next;
268                 } else {
269                     auto tmp = lastElSwapped->next;
270                     lastElSwapped->next = i->next;
271                     i->next = i->next->next;
272                     lastElSwapped = lastElSwapped->next;
273                     lastElSwapped->next = tmp;
274                 }
275             }
276 
277             i = i->next;
278         }
279 
280         return lastElSwapped;
281     }
282 
283 }
284 
285 template <typename T> void TList<T>::sort()
286 {
287     head->next = psort(head->next);
288 }
```

```
289
290 template <typename T> void TList<T>::parSort()
291 {
292     head->next = pparsort(head->next);
293 }
294
295 template <typename T> auto TList<T>::pparsort(std::shared_ptr<TNode>& head)
296 {
297     if (head == nullptr || head->next == nullptr) {
298         return head;
299     }
300
301     auto partitionedEl = partition(head);
302     auto leftPartition = partitionedEl->next;
303     auto rightPartition = head;
304
305     partitionedEl->next = nullptr;
306
307     if (leftPartition == nullptr) {
308         leftPartition = head;
309         rightPartition = head->next;
310         head->next = nullptr;
311     }
312
313     std::packaged_task<std::shared_ptr<TNode>(std::shared_ptr<TNode>&)>
314         task1(std::bind(&TList<T>::pparsort, this, std::placeholders::_1));
315     std::packaged_task<std::shared_ptr<TNode>(std::shared_ptr<TNode>&)>
316         task2(std::bind(&TList<T>::pparsort, this, std::placeholders::_1));
317     auto rightPartitionHandle = task1.get_future();
318     auto leftPartitionHandle = task2.get_future();
319
320
321     std::thread(std::move(task1), std::ref(rightPartition)).join();
322     rightPartition = rightPartitionHandle.get();
323     std::thread(std::move(task2), std::ref(leftPartition)).join();
324     leftPartition = leftPartitionHandle.get();
325     auto iter = leftPartition;
326     while (iter->next != nullptr) {
327         iter = iter->next;
328     }
329
330     iter->next = rightPartition;
331
332     return leftPartition;
333 }
334
335
336 #endif
337 //main.cpp
```

```
338 #include "TList.h"
339 #include <iostream>
340 #include "trapeze.h"
341 #include "square.h"
342 #include "rectangle.h"
343 #include "TStack.h"
344
345 void Display(void)
346 {
347     std::cout << "Display:" << std::endl;
348     std::cout << "1) Add Trapezoid" << std::endl;
349     std::cout << "2) Add Rectangle" << std::endl;
350     std::cout << "3) Add Square" << std::endl;
351     std::cout << "4) Delete" << std::endl;
352     std::cout << "5) Print" << std::endl;
353     std::cout << "0) Exit" << std::endl;
354 }
355
356 int main(void)
357 {
358     //
359     TStack<TList<Figure>, std::shared_ptr<Figure> > stack;
360     int act, index;
361     do {
362         Display();
363         std::cin >> act;
364         system("clear");
365         switch(act) {
366             case 1:
367
368                 stack.Push(std::make_shared<Trapezoid>(std::cin));
369                 break;
370             case 2:
371
372                 stack.Push(std::make_shared<Rectangle>(std::cin));
373                 break;
374             case 3:
375                 stack.Push(std::make_shared<Square>(std::cin));
376                 break;
377             case 4: {
378                 std::cout << "Enter principle of removal" << std::endl;
379                 std::cout << "1) by type" << std::endl;
380                 std::cout << "2) lesser than square" << std::endl;
381                 std::cin >> index;
382                 switch (index) {
383                     case 1: {
384                         std::cout << "Enter type" << std::endl;
385                         std::cout << "1) Trapezoid" << std::endl;
386                         std::cout << "2) Rectangle" << std::endl;
```

```
387                         std::cout << "3) Square" << std::endl;
388                         std::cin >> index;
389                         stack.RemoveByType(index);
390                         break;
391                     }
392                     case 2: {
393                         double area;
394                         std::cout << "Enter square" << std::endl;
395                         std::cin >> area;
396                         stack.RemoveBySquare(area);
397                         break;
398                     }
399                     default: {std::cout << "Unknown command\n"; break;}
400                 }
401                 break;
402             }
403             case 5:
404                 stack.Print();
405                 break;
406             case 0:
407                 break;
408             default:
409                 std::cout << "Incorrect command" << std::endl;;
410                 break;
411         }
412     } while(act);
413 
414     return 0;
415 }
```

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.