**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 6**

Тема: Основы работы с коллекциями, аллокаторы

Студент: Ефимов Александр

Группа: 80-201

Преподаватель: Чернышов Л.Н.

Дата: 29.11.2019

Оценка:

Москва, 2019

1. **Постановка задачи**

Создать шаблон динамической коллекцию, согласно варианту задания:

* Коллекция должна быть реализована с помощью умных указателей (std::shared\_ptr, std::weak\_ptr).

(Опционально использование std::unique\_ptr;)

* В качестве параметра шаблона коллекция должна принимать тип данных;
* Коллекция должна содержать метод доступа:
  + Стек – pop, push, top;
  + Очередь – pop, push, top;
  + Список, Динамический массив – доступ к элементу по оператору [];
* Реализовать аллокатор, который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (Динамический массив, Список, Стек, Очередь);
* Коллекция должна использовать аллокатор для выделеления и освобождения памяти для своих элементов.
* Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).
* Реализовать программу, которая:
  + Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;
  + Позволяет удалять элемент из коллекции по номеру элемента;
  + Выводит на экран введенные фигуры c помощью std::for\_each;

Вариант 2: *Квадрат, Стек, Список*.

1. **Описание программы**

Шаблон квадрата написан в *Shape.h*, вместе шаблонными функциями *Area*, *Centre* и *Print*, которые ищут его площадь, центр фигуры и печатают его соответственно.

Шаблон стека расписан в *Stack.h*. Он выполняет все стандартные для стека функции (*pop, push, top*). В нем также расписан собственный итератор *forward\_iterator* для шаблонных функций, работающих с итераторами (*std::count\_if, std::for\_each*). В шаблон передается тип аллокатора, по умолчанию берется **std::allocator**.

В *Allocator.h* содержится аллокатор, минимально соответствующий чертам аллокатора, заданым в стандарте. В нем реализованы стандартный конструктор и конструктор из аллокатора, шаблонная структура *rebind*, позволяющая перешивать аллокатор на другой тип, методы *allocate* и *deallocate*, выделяющие и отпускающие память, *contruct* и *destroy*, создающие и уничтожающие типы.

В *main.cpp* содержится меню, позволяющее работать со стеком, держащий в себе тип *Square*, тип точек которого **int**.

1. **Набор testcases**

* **Test\_01.cpp**

Использование *pop*, *push* и *top*.

* **Test\_02.cpp**

Использование функций аллокатора с типом **int**

* **Test\_03.cpp**

Проверка работы с **std::map** и **std::list**.

* **Test\_04.txt**

Показание возможностей меню *main.cpp*

1. **Результаты выполнения тестов**

Все тесты возвращают ожидаемые результаты

**test\_01**

|  |
| --- |
| Current stack address: 0  Attempting to perform pop...  Stack is empty  Pushing 5...  Current address 0x558439948f50 with size of 1 and top element 5  Pushing 1, 7, 7, 0, 1, 3...  Current address 0x558439948f50 with size of 7 and top element 3  Poping one element...  Current address 0x558439948f50 with size of 6 and top element 1 |

**test\_02**

|  |
| --- |
| Created pointer p: 0  Allocated to pointer p: 0x5580a40a1ea8  Constructed in pointer p1 (0x5580a40a1ea8) with value 10  Created p2, allocated (0x5580a40a1ea4) and constructed with value 20  Destroyed and deallocated pointer p1  Destroyed and deallocated pointer p2 |

**test\_03**

|  |
| --- |
| Allocated map:  0 1  1 1  2 2  3 6  4 24  5 120  6 720  7 5040  8 40320  9 362880  Allocated list:  1  1  2  6  24  120  720  5040  40320  362880 |

**test\_04**

|  |
| --- |
| 1. Add square  2. Print all elements  3. Amount of squares, which have area less than  4. Print this menu  5. Delete element  6. Delete all elements  0. Exit  ---------------  Menu choice: 1  Input point coordinates clockwise or counter clockwise  A: 0 0  B: 0 4  C: 4 4  D: 4 0  ---------------  Menu choice: 1  Input point coordinates clockwise or counter clockwise  A: -6 10  B: -5 6  C: -9 5  D: -10 9  ---------------  Menu choice: 1  Input point coordinates clockwise or counter clockwise  A: 0 -2  B: 5 3  C: -7 15  D: -12 10  ---------------  Menu choice: 2  1. Square: (0 , 0); (0 , 4); (4 , 4); (4 , 0); Centre = (2 , 2); Area = 16  2. Square: (-6 , 10); (-5 , 6); (-9 , 5); (-10 , 9); Centre = (-7.5 , 7.5); Area = 17  3. Abstract: (0 , -2); (5 , 3); (-7 , 15); (-12 , 10); Centre = (-3.5 , 6.5); Area = 120  ---------------  Menu choice: 3  Max area: 16  The amount of squares that are equal to or lower than 16 is 1  ---------------  Menu choice: 3  Max area: 17  The amount of squares that are equal to or lower than 17 is 2  ---------------  Menu choice: 3  Max area: 125  The amount of squares that are equal to or lower than 125 is 2  ---------------  Menu choice: 3  Max area: 10  The amount of squares that are equal to or lower than 10 is 0  ---------------  Menu choice: 5  Number of the square: 1  ---------------  Menu choice: 5  Number of the square: 2  ---------------  Menu choice: 2  1. Square: (-6 , 10); (-5 , 6); (-9 , 5); (-10 , 9); Centre = (-7.5 , 7.5); Area = 17  ---------------  Menu choice: 6  Stack erased  ---------------  Menu choice: 2  The stack is currently empty  ---------------  Menu choice: |

1. **Листинг программы**

**Shape.h**

|  |
| --- |
| #ifndef SHAPE\_H  #define SHAPE\_H  #include <iostream>  #include <utility>  #include <cmath>  typedef std::pair<double, double> DoublePoint;  template <typename T>  class Square {  public:  typedef std::pair<T, T> Point;  Square() : centre(DoublePoint(0,0)), area(0)  {  for (int i = 0; i < 4; ++i)  p[i].first = p[i].second = 0;  }  bool abstract = false;  std::pair<double, double> centre;  double area{ 0 };  Point p[4];  };  //-----------------------------------------------------------------------------------------------------------  bool DoubleEqual(double lhs, double rhs)  {  const double EPS = 0.00001;  if (lhs > rhs)  return (lhs - rhs) < EPS ? true : false;  else  return (rhs - lhs) < EPS ? true : false;  }  bool IsRight(std::pair<double, double> a, std::pair<double, double> b, std::pair<double, double> c)  {  std::pair<double, double> vec1 = { b.first - a.first, b.second - a.second };  std::pair<double, double> vec2 = { c.first - a.first, c.second - a.second };  double result = vec1.first \* vec2.first + vec1.second \* vec2.second;  if (DoubleEqual(result, 0)) return true;  return false;  }  bool IsRight(std::pair<int, int> a, std::pair<int, int> b, std::pair<int, int> c)  {  std::pair<int, int> vec1 = { b.first - a.first, b.second - a.second };  std::pair<int, int> vec2 = { c.first - a.first, c.second - a.second };  int result = vec1.first \* vec2.first + vec1.second \* vec2.second;  if (result == 0) return true;  return false;  }  //-----------------------------------------------------------------------------------------------------------  template <class T>  bool IsRectangle(std::pair<T, T> p[4])  {  std::pair<T, T> null(0, 0);  if (p[0] == null && p[1] == null && p[2] == null && p[3] == null) return false;  if (  IsRight(p[2], p[1], p[3]) &&  IsRight(p[3], p[2], p[0]) &&  IsRight(p[1], p[0], p[2]) &&  IsRight(p[0], p[3], p[1])  ) return true;  return false;  }  //-----------------------------------------------------------------------------------------------------------  template <class T>  inline double distance(std::pair<T, T> a, std::pair<T, T> b)  {  return (b.first - a.first) \* (b.first - a.first) + (b.second - a.second) \* (b.second - a.second);  }  template <class T>  bool IsSquare(std::pair<T, T> p[4])  {  if (  IsRectangle(p) &&  DoubleEqual(distance(p[0], p[1]), distance(p[1], p[2])) &&  DoubleEqual(distance(p[1], p[2]), distance(p[2], p[3])) &&  DoubleEqual(distance(p[2], p[3]), distance(p[3], p[0])) &&  DoubleEqual(distance(p[3], p[0]), distance(p[0], p[1]))  ) return true;  return false;  }  //-----------------------------------------------------------------------------------------------------------  template <class T>  DoublePoint Centre(Square<T>& sqr)  {  DoublePoint res(0, 0);  for (int i = 0; i < 4; ++i)  {  res.first += sqr.p[i].first;  res.second += sqr.p[i].second;  }  res.first /= 4;  res.second /= 4;  return res;  }  template <class T>  double Area(Square<T>& sqr)  {  return sqrt(distance(sqr.p[0], sqr.p[1]) \* distance(sqr.p[1], sqr.p[2]));  }  template <class T>  std::ostream& operator << (std::ostream& os, const std::pair<T, T>& p)  {  os << '(' << p.first << " , " << p.second << ')';  return os;  }  template <class T>  std::ostream& operator << (std::ostream& os, Square<T>& sqr)  {  if (sqr.abstract)  os << "Abstract: ";  else  os << "Square: ";  os << sqr.p[0] << "; " << sqr.p[1] << "; " << sqr.p[2] << "; " << sqr.p[3] << "; Centre = " << sqr.centre << "; Area = " << sqr.area;  return os;  }  //-----------------------------------------------------------------------------------------------------------  bool operator == (std::pair<int, int> lhs, std::pair<int, int> rhs)  {  return lhs.first == rhs.first && lhs.second == rhs.second;  }  bool operator == (std::pair<double, double> lhs, std::pair<double, double> rhs)  {  return DoubleEqual(lhs.first, rhs.first) && DoubleEqual(lhs.second, rhs.second);  }  template <class T>  std::istream& operator >> (std::istream& is, Square<T>& sqr)  {  std::cout << "Input point coordinates clockwise or counter clockwise\nA: ";  is >> sqr.p[0].first >> sqr.p[0].second;  std::cout << "B: ";  is >> sqr.p[1].first >> sqr.p[1].second;  std::cout << "C: ";  is >> sqr.p[2].first >> sqr.p[2].second;  std::cout << "D: ";  is >> sqr.p[3].first >> sqr.p[3].second;  if (IsSquare(sqr.p)) sqr.abstract = false;  else sqr.abstract = true;  sqr.centre = Centre(sqr);  sqr.area = Area(sqr);  return is;  }  #endif |

**Stack.h**

|  |
| --- |
| #ifndef STACK\_H  #define STACK\_H  #include <memory>  #include <exception>  #include <algorithm>  template <  class T,  class Alloc = std::allocator<T>  > class Stack {  public:  class forward\_iterator;  class StackNode;  Stack()  {  container = nullptr;  \_size = 0;  }  ~Stack() { erase(); }  size\_t size(void) const  {  return \_size;  }  bool empty(void) const  {  return size() == 0;  }  forward\_iterator begin(void) const  {  return forward\_iterator(container);  }  forward\_iterator end(void) const  {  return forward\_iterator();  }  Stack& push(T value)  {  insert(\_size, value);  return \*this;  }  Stack& pop(void)  {  if (!empty())  erase(\_size - 1);  else throw std::logic\_error("Stack is empty");  return \*this;  }  T top(void)  {  if (!empty())  return container.get()->top();  else throw std::logic\_error("Stack is empty");  }  void insert(size\_t index, T val)  {  if (index == 0)  {  container = fabricateNode(val, container);  }  else if (index > 0 && index <= \_size)  {  --index;  forward\_iterator begin = Stack<T, Alloc>::begin();  for (forward\_iterator end = Stack<T, Alloc>::end(); index != 0; ++begin)  {  --index;  }  std::shared\_ptr<StackNode> allocated = fabricateNode(val, begin.value.lock().get()->next);  begin.value.lock().get()->insert(std::move(allocated));  }  else throw std::logic\_error("Outside the bounds");    ++\_size;  }  void erase(size\_t index)  {  if (container == nullptr) throw std::logic\_error("Stack is empty");  if (index == 0)  {  container = container.get()->next;  }  else if (index > 0 && index < \_size)  {  --index;  forward\_iterator begin = Stack<T, Alloc>::begin();  for (forward\_iterator end = Stack<T, Alloc>::end(); index != 0; ++begin)  {  --index;  }  begin.value.lock().get()->erase();  }  else throw std::logic\_error("Outside the bounds");  --\_size;  }  void erase()  {  container = nullptr;  \_size = 0;  }  private:  std::shared\_ptr<StackNode> container;  size\_t \_size;  std::shared\_ptr<StackNode> fabricateNode (T val, const std::shared\_ptr<StackNode> &oneAfterNext)  {  StackNode \*raw = allocator.allocate(1);  allocator.construct(raw, val, oneAfterNext);  std::shared\_ptr<StackNode> result (  raw,  [this](StackNode \*p) {  p->next = nullptr;  allocator.deallocate(p, 1);  }  );  return result;  }  using AllocNode = typename Alloc::template rebind<StackNode>::other;  AllocNode allocator;  };  template <class T, class Alloc>  class Stack<T, Alloc>::forward\_iterator {  public:  //The next using's actually make our pointer compatible with pointer templates  using iterator\_category = std::forward\_iterator\_tag;  using difference\_type = std::ptrdiff\_t;  using value\_type = T ;  using pointer = T\*;  using reference = T&;  forward\_iterator() { value = std::shared\_ptr<StackNode> (nullptr); }  forward\_iterator(const std::shared\_ptr<StackNode> &arg) { value = arg; }  ~forward\_iterator() { };  forward\_iterator& operator = (const std::shared\_ptr<StackNode> &arg)  {  value = arg;  return \*this;  }  forward\_iterator& operator ++ ()  {  if (value.expired()) return \*this;  value = value.lock().get()->next;  return \*this;  }  forward\_iterator operator ++ (int)  {  if (value.expired()) return \*this;  forward\_iterator prev(value.lock());  value = value.lock().get()->next;  return prev;  }  T& operator \*()  {  if (value.expired()) throw std::logic\_error("Invalid iterator");  return value.lock().get()->data;  }  bool operator == (const forward\_iterator& other) const  {  return value.lock() == other.value.lock();  }  bool operator != (const forward\_iterator& other) const  {  return !(\*this == other);  }  std::weak\_ptr<StackNode> value;  };  template <class T, class Alloc>  class Stack<T, Alloc>::StackNode {  friend Stack<T, Alloc>;  public:  StackNode(T val, const std::shared\_ptr<StackNode> &oneAfterNext) // If we add in the middle, oneAfterNext will be element after out new one  {  data = val;  if (oneAfterNext.get() != nullptr)  next = oneAfterNext;  }  T top(void)  {  if (next.use\_count() != 0) {  return next.get()->top();  }  else return data;  }    void insert(std::shared\_ptr<StackNode> Allocated)  {  next = Allocated;  }  void erase()  {  next = next.get()->next;  }    private:  T data;  std::shared\_ptr<StackNode> next;  };  #endif |

**Allocator.h**

|  |
| --- |
| #ifndef ALLOCATOR\_H  #define ALLOCATOR\_H  #include <exception>  #include <memory>  template <class T, size\_t BLOCK\_COUNT>  class Allocator {  public:  using value\_type = T;  using pointer = value\_type\*;  using reference = value\_type&;  using const\_reference = const value\_type&;  using size\_type = size\_t;  using difference\_type = ptrdiff\_t;  size\_type getFreeSize (void) const { return freeCount; }  char \* getAllocBlocks(void) const { return allocatedBlocks; }  void \*\* getFreeBlocks (void) const { return freeBlockPoint; }  Allocator()  {  freeCount = BLOCK\_COUNT;  allocatedBlocks = static\_cast<char\* >(operator new(BLOCK\_COUNT \* sizeof(T) ));  freeBlockPoint = static\_cast<void\*\* >(operator new(BLOCK\_COUNT \* sizeof(void\*) ));  for (size\_type i = 0; i < BLOCK\_COUNT; ++i)  {  freeBlockPoint[i] = allocatedBlocks + i \* sizeof(T);  }  }  template<typename U>  Allocator(const Allocator<U, BLOCK\_COUNT> &other)  {  freeCount = other.getFreeSize();  allocatedBlocks = other.getAllocBlocks();  freeBlockPoint = other.getFreeBlocks();  }  ~Allocator()  {  operator delete(allocatedBlocks);  operator delete(freeBlockPoint);  }  template<class U>  struct rebind {  using other = Allocator<U, BLOCK\_COUNT>;  };  pointer allocate(size\_type amount = 1)  {  if (freeCount == 0) throw std::bad\_alloc();  return (pointer) freeBlockPoint[--freeCount];  }    void deallocate(pointer ptr, size\_type amount = 1)  {  size\_type insertIndex;  for (insertIndex = 0; insertIndex < freeCount; ++insertIndex)  {  if (ptr < freeBlockPoint[insertIndex]) break;  }  for (size\_type i = freeCount; i != insertIndex; --i)  {  freeBlockPoint[i] = freeBlockPoint[i - 1];  }  freeBlockPoint[insertIndex] = ptr;  ++freeCount;  }  template<typename U, typename... Args>  void construct(U \*ptr, Args &&...args)  {  new (ptr) U (std::forward<Args>(args)...);  }  void destroy(pointer ptr)  {  ptr->~T();  }  private:  size\_type freeCount;  void \*\*freeBlockPoint;  char \*allocatedBlocks;  };  template<typename T1, size\_t SZ1, typename T2, size\_t SZ2>  inline bool operator == (const Allocator<T1, SZ1>&, const Allocator<T2, SZ2>&)  { return true; }  template<typename T, size\_t SZ>  inline bool operator == (const Allocator<T, SZ>&, const Allocator<T, SZ>&)  { return true; }  template<typename T1, size\_t SZ1, typename T2, size\_t SZ2>  inline bool operator != (const Allocator<T1, SZ1>&, const Allocator<T2, SZ2>&)  { return false; }  template<typename T, size\_t SZ>  inline bool operator != (const Allocator<T, SZ>&, const Allocator<T, SZ>&)  { return false; }  #endif |

**main.cpp**

|  |
| --- |
| #include <iostream>  #include <string>  #include <exception>  #include "Stack.h"  #include "Shape.h"  using namespace std;  struct stack\_empty : public std::exception  {  const char\* what() const throw ()  {  return "The stack is currently empty";  }  };  void PrintStack(const Stack< Square<int> >& stack)  {  int num = 1;  for\_each(stack.begin(), stack.end(), [&num](auto& e)  {  cout << num << ". " << e << endl;  ++num;  });  }  int GetNum(void)  {  int k;  bool got = false;  while (!got)  {  cin >> k;  if (!cin.good())  {  cout << "Bad input ignored: " << flush;  cin.clear();  cin.ignore(256, '\n');  }  else got = true;  }  return k;  }  int main()  {  Stack< Square<int> > SquareStack;  Square<int> buff;  int k = 1;  size\_t num;  double MaxArea;  cout  << "\n1. Add square\n"  << "2. Print all elements\n"  << "3. Amount of squares, which have area less than\n"  << "4. Print this menu\n"  << "5. Delete element\n"  << "6. Delete all elements\n"  << "0. Exit\n"  << flush;  do  {  cout << "---------------\nMenu choice: ";  try  {  k = GetNum();  if (!cin) throw runtime\_error("Failed to get input");  switch (k)  {  case 1:  cin >> buff;  SquareStack.push(buff);  k = 1;  break;  case 2:  if (SquareStack.empty()) throw stack\_empty();  PrintStack(SquareStack);  break;  case 3:  if (SquareStack.empty()) throw stack\_empty();  cout << "Max area: ";  cin >> MaxArea;  cout << "The amount of squares that are equal or lower is " << count\_if( SquareStack.begin(), SquareStack.end(), [&MaxArea](auto& sqr) -> bool  {  return ( !sqr.abstract && (sqr.area < MaxArea || DoubleEqual(sqr.area, MaxArea)) );  }  ) << endl;  break;  case 4:  cout  << "1. Add square\n"  << "2. Print all elements\n"  << "3. Amount of squares, which have area less than\n"  << "4. Print this menu\n"  << "5. Delete element\n"  << "6. Delete all elements\n"  << "0. Exit\n"  << endl;  break;  case 5:  if (SquareStack.empty()) throw stack\_empty();  cout << "Number of the square: ";  num = GetNum();  if (num < 1 || num > SquareStack.size()) throw runtime\_error("Incorrect index");  SquareStack.erase(num - 1);  break;  case 6:  SquareStack.erase();  cout << "Stack erased" << endl;  break;  case 0:  break;  default:  cout << "No such number" << endl;  }  }  catch (stack\_empty& err)  {  cout << err.what() << endl;  }  catch (runtime\_error& err)  {  cerr << err.what() << endl;  }  catch (logic\_error & err)  {  cerr << "In stack: " << err.what() << endl;  }  catch (...)  {  cerr << "Exception thrown, but undefined. Please open an issue at GitHub with steps to reproduce ." << endl;  }  } while (k);  return 0;  } |

1. **Выводы**

Реализация собственного аллокатора позволяет, например, избавится от постоянных выделений на память, заранее сделав выделение некоторой области памяти и работая только с ней, что ускоряет программу за счет работы только уже с выделенной памятью, но ограничивает объем используемой памяти до заданной (что может быть плюсом в системах с большой нагрузкой на память, не требующих скорости).

1. **Список литературы**

* Черты аллокатора (Справочник по языке C++ )[Электронный ресурс]. URL: <https://en.cppreference.com/w/cpp/named_req/Allocator> (дата обращения: 23.11.2019).
* Метод make\_shared (Справочник по языке C++ )[Электронный ресурс].   
  URL: <https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared> (дата обращения: 23.11.2019).