Московский авиационный институт
(Национальный исследовательский университет)

Институт: «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: Бирюков В. В.
Группа: 80-207
Преподаватель: Чернышов Л.Н.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

Создать шаблон динамической коллекцию, согласно варианту задания:

1. Коллекция должна быть реализована с помощью умных указателей (std::shared_ptr, std::weak_ptr). Опционально использование std::unique_ptr.
2. В качестве параметра шаблона коллекция должна принимать тип данных - фигуры.
3. Коллекция должна содержать метод доступа:
   - стек – pop, push, top.
   - очередь – pop, push, top.
   - список, Динамический массив – доступ к элементу по оператору [].
4. 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);
5. Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.
6. Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).
7. 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 с помощью std::for_each.

Вариант 17: треугольник, очередь, динамический массив.

2. Описание программы

Шаблонный класс Triangle хранит треугольник как координаты его левой вершины и длину стороны.

Шаблонный класс Queue представляет собой коллекцию типа “очередь”. Очередь хранит элементы в виде линейного двусвязного списка.

Аллокатор представляет из себя простейший линейный аллокатор, который хранит указатель на всю выделенную память, адреса свободных блоков в динамическом массиве и индекс первого свободного блока. При

выделении памяти этот индекс сдвигается на нужную величину. В силу способа хранения, переиспользование памяти невозможно. Аллокатор содержит все необходимые методы. Аллокатор совместим со стандартными контейнерами std::list, std::map и std::vector.

3. Набор тестов

Программа принимает на вход команды, вводимые пользователем. Реализованы следующие команды: добавление элемента в начало очереди (`a TRIANGLE`), удаление элемента из конца очереди (`d`), просмотр элемента в конце очереди (`t`), вставка элемента по индексу (`i INDEX TRIANGLE`), удаление элемента по индексу (`e INDEX`), печать элементов очереди (`p`).

Треугольник вводится в виде трех чисел: координат вершины и длины стороны.

Тест 1:

```
Allocating 28 B, 168 B left
> a 1 1 1
Allocating 28 B, 140 B left
> a 2 2 2
Allocating 28 B, 112 B left
> a 3 3 3
Allocating 28 B, 84 B left
> p
(3, 3) (4.5, 5.59808) (6, 3)
(2, 2) (3, 3.73205) (4, 2)
(1, 1) (1.5, 1.86603) (2, 1)
> d
Deallocating
> d
Deallocating
> p
(3, 3) (4.5, 5.59808) (6, 3)
> a 1 1 1
Allocating 28 B, 56 B left
> p
(1, 1) (1.5, 1.86603) (2, 1)
(3, 3) (4.5, 5.59808) (6, 3)
> i 1 2 2 2
Allocating 28 B, 28 B left
> p
(1, 1) (1.5, 1.86603) (2, 1)
(2, 2) (3, 3.73205) (4, 2)
(3, 3) (4.5, 5.59808) (6, 3)
> e 0
Deallocating
```

```
> p
(2, 2) (3, 3.73205) (4, 2)
(3, 3) (4.5, 5.59808) (6, 3)
> t
(3, 3) (4.5, 5.59808) (6, 3)
> a 6 6 6
Allocating 28 B, 0 B left
> p
(6, 6) (9, 11.1962) (12, 6)
(2, 2) (3, 3.73205) (4, 2)
(3, 3) (4.5, 5.59808) (6, 3)
> q
Deallocating
Deallocating
Deallocating
Deallocating
```

Тест 2 (исключительные ситуации):

```
Allocating 28 B, 168 B left
> p
> t
Error: Queue is empty
> d
Error: Queue is empty
> a 1 6 0
Error: Invalid triangle parameters
> a 1 1 1
Allocating 28 B, 140 B left
> a 2 2 2
Allocating 28 B, 112 B left
> a 3 3 3
Allocating 28 B, 84 B left
> p
(3, 3) (4.5, 5.59808) (6, 3)
(2, 2) (3, 3.73205) (4, 2)
(1, 1) (1.5, 1.86603) (2, 1)
> i -1 9 9 9
Error: Out of bounds
> e 5
Error: Out of bounds
> p
(3, 3) (4.5, 5.59808) (6, 3)
(2, 2) (3, 3.73205) (4, 2)
(1, 1) (1.5, 1.86603) (2, 1)
> a 4 4 4
Allocating 28 B, 56 B left
> a 5 5 5
Allocating 28 B, 28 B left
> a 6 6 6
Allocating 28 B, 0 B left
```

```
> a 7 7 7
std::bad_alloc
> p
(6, 6) (9, 11.1962) (12, 6)
(5, 5) (7.5, 9.33013) (10, 5)
(4, 4) (6, 7.4641) (8, 4)
(3, 3) (4.5, 5.59808) (6, 3)
(2, 2) (3, 3.73205) (4, 2)
(1, 1) (1.5, 1.86603) (2, 1)
> q
Deallocating
Deallocating
Deallocating
Deallocating
Deallocating
Deallocating
Deallocating
```

4. Листинг программы

```cpp
// allocator.hpp
// Линейный аллокатор. Хранит адреса свободных блоков в массиве

#pragma once

#include <vector>

template <class T, size_t BLOCK_SIZE>
class Allocator {
private:
  T* buffer;
  std::vector<T*> free_blocks;
  size_t last;

public:
  using value_type = T;
  using pointer = T * ;
  using const_pointer = const T*;
  using size_type = std::size_t;

  Allocator(const Allocator<T, BLOCK_SIZE> & other) :
                                                Allocator() {
    last = other.last;
    for (size_t i = 0; i < BLOCK_SIZE; ++i) {
      buffer[i] = other.buffer[i];
      free_blocks[i] = &buffer[i];
    }
  }

  Allocator() {
    static_assert(BLOCK_SIZE > 0,
                  "Cannot create empty allocator");
    buffer = new T[BLOCK_SIZE];
    free_blocks.resize(BLOCK_SIZE);
```

```
    for (size_type i = 0; i < BLOCK_SIZE; ++i) {
      free_blocks[i] = &buffer[i];
    }
    last = 0;
  }

  ~Allocator() {
    delete[] buffer;
  }

  template <class U>
  struct rebind {
    using other = Allocator<U, BLOCK_SIZE>;
  };

  pointer allocate(size_type n) {
    if (last + n >= free_blocks.size()) {
      throw std::bad_alloc();
    }
    pointer ptr = free_blocks[last];
    last += n;

    std::cout << "Allocating " << sizeof(T) << " B, "
              << (free_blocks.size() - last - 1) * sizeof(T)
              << " B left" << std::endl;

    return ptr;
  }

  void deallocate(pointer, size_type) {
    std::cout << "Deallocating\n";
  }
};

// triangle.hpp
// Треугольник. Хранит данные как координаты левой вершины и длину
стороны.

#pragma once

#include <utility>
#include <stdexcept>
#include <cmath>

template <class T>
class Triangle {
public:
  std::pair<T,T> x;
  T a;

  Triangle() : x(), a() {}
  Triangle(T x1, T x2, T a) : x(x1,x2), a(a) {
    if (a <= 0) {
      throw std::invalid_argument("Error: Invalid triangle
                                   parameters");
    }
```

```
  }

  double square() const {
    return sqrt(3) / 4 * a * a;
  }
  template <class A>
  friend std::ostream& operator<<(std::ostream&,
                                  const Triangle<A>&);
  template <class A>
  friend std::istream& operator>>(std::istream&,
                                  const Triangle<A>&);
};

template <class T>
std::ostream& operator<<(std::ostream& os,
                         const Triangle<T>& tr) {
  os << "(" << tr.x.first << ", " << tr.x.second << ") "
     << "(" << tr.x.first + 1.0 / 2 * tr.a << ", "
     << tr.x.second + sqrt(3) / 2 * tr.a << ") "
     << "(" << tr.x.first + tr.a << ", " << tr.x.second << ")";
  return os;
}

template <class T>
std::istream& operator>>(std::istream& is, Triangle<T>& tr) {
  is >> tr.x.first >> tr.x.second >> tr.a;
  if (tr.a <= 0) {
    throw std::invalid_argument("Error: Invalid triangle
                                 parameters");
  }
  return is;
}

// queue.hpp
// Структура данных очередь. Реализована на двусвязном списке.

#pragma once

#include <memory>
#include <stdexcept>

template <class T, class ALLOCATOR = std::allocator<T>>
class Queue {
private:
  class Node {
  public:
    using allocator_type = typename ALLOCATOR::template
                                    rebind<Node>::other;

    static allocator_type& get_allocator() {
      static allocator_type allocator;
      return allocator;
    }

    struct deleter_type {
      deleter_type() = default;
```

```
      void operator() (Node* ptr) {
        get_allocator().deallocate(ptr, 1);
      }
    };

    static deleter_type deleter;

    T data;
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;

    Node() : data(), next(), prev() {}
    Node(const T& value) : data(value), next(), prev() {}
  };

public:
  class Forward_iterator {
  private:
    std::shared_ptr<Node> ptr;

  public:
    using iterator_category = std::forward_iterator_tag;
      using value_type = T;
      using difference_type = size_t;
      using pointer = T*;
      using reference = T&;
    T& operator*() {
      return ptr->data;
    }
    Forward_iterator& operator++() {
      if (ptr == nullptr) {
        throw std::runtime_error("Error: Out of bounds");
      }
      ptr = ptr->next;
      return *this;
    }
    bool operator!=(const Forward_iterator& other) {
      return ptr != other.ptr;
    }
    bool operator==(const Forward_iterator& other) {
      return ptr == other.ptr;
    }
    Forward_iterator(std::shared_ptr<Node> ptr) : ptr(ptr){}

    friend class Queue<T, ALLOCATOR>;
  };

private:
  std::shared_ptr<Node> head;
  std::shared_ptr<Node> tail;

public:
  Forward_iterator begin();
  Forward_iterator end();
  void insert(Forward_iterator&, const T&);
  void erase(Forward_iterator&);
```

```
  const T& top();
  void pop();
  void push(const T&);

  Queue();
};

template <class T, class ALLOCATOR>
Queue<T, ALLOCATOR>::Queue() {
  Node* ptr = Node::get_allocator().allocate(1);
  tail = std::shared_ptr<Node>(new (ptr) Node, Node::deleter);
  head = tail;
}

template <class T, class ALLOCATOR>
const T& Queue<T, ALLOCATOR>::top() {
  if (tail->prev.lock() == nullptr) {
    throw std::runtime_error("Error: Queue is empty");
  }
  return tail->prev.lock()->data;
}

template <class T, class ALLOCATOR>
void Queue<T, ALLOCATOR>::pop() {
  if (tail->prev.lock() == nullptr) {
    throw std::runtime_error("Error: Queue is empty");
  }
  tail = tail->prev.lock();
  tail->next = nullptr;
  if (head == tail) {
    head->next = nullptr;
  }
}

template <class T, class ALLOCATOR>
void Queue<T, ALLOCATOR>::push(const T &value) {
  Node* ptr = Node::get_allocator().allocate(1);
  std::shared_ptr<Node> node(new (ptr) Node(value), Node::deleter);
  node->next = head;
  head->prev = node;
  head = node;
}

template <class T, class ALLOCATOR>
typename Queue<T, ALLOCATOR>::Forward_iterator
Queue<T, ALLOCATOR>::begin() {
  return Forward_iterator(head);
}

template <class T, class ALLOCATOR>
typename Queue<T, ALLOCATOR>::Forward_iterator
Queue<T, ALLOCATOR>::end() {
  return Forward_iterator(tail);
}

template <class T, class ALLOCATOR>
```

```
void Queue<T, ALLOCATOR>::insert(typename Queue<T,
ALLOCATOR>::Forward_iterator& iter, const T& value) {
  if (iter.ptr == nullptr) {
    throw std::runtime_error("Error: Out of bounds");
  }
  if (iter == begin()) {
    push(value);
  } else {
    Node* ptr = Node::get_allocator().allocate(1);
    std::shared_ptr<Node> node(new (ptr) Node(value),
                               Node::deleter);
    node->next = iter.ptr;
    node->prev = iter.ptr->prev;
    iter.ptr->prev.lock()->next = node;
    iter.ptr->prev = node;
  }
}

template <class T, class ALLOCATOR>
void Queue<T, ALLOCATOR>::erase(typename Queue<T,
ALLOCATOR>::Forward_iterator& iter) {
  if (iter.ptr == nullptr || iter == end()) {
    throw std::runtime_error("Error: Out of bounds");
  }
  if (iter == begin()) {
    head = head->next;
  } else {
    iter.ptr->prev.lock()->next = iter.ptr->next;
  }
}

// main.cpp
#include <iostream>
#include <algorithm>

#include "queue.hpp"
#include "triangle.hpp"
#include "allocator.hpp"

int main() {
  Queue<Triangle<int>, Allocator<Triangle<int>,8>> queue;

  std::cout << "a TRIANGLE - Push" << std::endl
            << "d - Pop" << std::endl
            << "t - Top" << std::endl
            << "i INDEX TRIANGLE - Insert" << std::endl
            << "e INDEX - Erase" << std::endl
            << "p - Print" << std::endl
            << "q - Exit" << std::endl;

  char command;
  std::cout << "> ";
  while (std::cin >> command && command != 'q') {
    try {
      if (command == 'a') {
        Triangle<int> tr;
```

```
      std::cin >> tr;
      queue.push(tr);

    } else if (command == 'd') {
      queue.pop();

    } else if (command == 't') {
      std::cout << queue.top() << std::endl;

    } else if (command == 'i') {
      int index;
      std::cin >> index;
      Triangle<int> tr;
      std::cin >> tr;
      if (index < 0) {
        throw std::runtime_error("Error: Out of bounds");
      }
      auto iter = queue.begin();
      while (index--) {
        ++iter;
      }
      queue.insert(iter, tr);

    } else if (command == 'e') {
      int index;
      std::cin >> index;
      if (index < 0) {
        throw std::runtime_error("Error: Out of bounds");
      }
      auto iter = queue.begin();
      while (index--) {
        ++iter;
      }
      queue.erase(iter);

    } else if (command == 'p') {
      std::for_each(queue.begin(), queue.end(),
                    [](const Triangle<int>& tr)
                    { std::cout << tr << std::endl; });
    }
  } catch (const std::exception& e) {
    std::cout << e.what() << std::endl;
  }
  std::cout << "> ";
 }
}
```

5. Выводы

В ходе лабораторной работы я познакомился с понятием аллокатора, а также с написанием собственного аллокатора, совместимого со стандартными контейнерами, и добавления поддержки аллокаторов в собственный контейнер.

Литература

1. Справочник по языку C++ [Электронный ресурс]. URL: https://ru.cppreference.com (дата обращения: 28.11.20).
2. Аллокаторы памяти [Электронный ресурс]. URL: https://habr.com/ru/post/505632/ (дата обращения: 28.11.20).