Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа
по курсу «ООП»

Тема:
Алллокатор .

Студент:	Касимов М.М.
Группа:	М80-206Б-18
Преподаватель:	Журавлев А.А.
Вариант:	6
Оценка:	
Дата:	

Москва
2019

1. Код программы на языке C++:

Pentagon.h:

```
#pragma once

#include "point.h"

template<class T>
struct pentagon {
private:
    point<T> a1,a2,a3,a4,a5;
public:
    point<T> center() const;
    void print(std::ostream& os) const ;
    double area() const;
    pentagon(std::istream& is);
};

template<class T>
double pentagon<T>::area() const {
    return  (0.5) * abs(((a1.x*a2.y + a2.x*a3.y + a3.x*a4.y + a4.x*a5.y + a5.x*a1.y) - (
a1.y*a2.x + a2.y*a3.x + a3.y*a4.x + a4.y*a5.x + a5.y*a1.x )));
}

template<class T>
pentagon<T>::pentagon(std::istream& is) {
    is >> a1 >> a2 >> a3 >> a4 >> a5;
}

template<class T>
void pentagon<T>::print(std::ostream& os) const {
    os << "coordinate:\n"<<"{\n"<< a1 << '\n' << a2 << '\n' << a3 << '\n' << a4 << '\n'
<< a5 << '\n' << "}\n";
}

template<class T>
point<T> pentagon<T>::center() const {
    T x,y;
    x = (a1.x + a2.x + a3.x + a4.x + a5.x) / 5;
    y = (a1.y + a2.y + a3.y + a4.y + a5.y) / 5;
    return {x,y};
}
```

Point.h:

```
#pragma once
```

```
#include <iostream>

template<class T>
struct point {
    T x;
    T y;
};

template<class T>
std::istream& operator>> (std::istream& is, point<T>& p) {
    is >> p.x >> p.y;
    return is;
}

template<class T>
std::ostream& operator<< (std::ostream& os, const point<T>& p) {
    os << p.x << ' ' << p.y;
    return os;
}
```

Mylist.h:

```
#pragma once

#include <iterator>
#include <memory>

namespace cntr {

    template<class T>
    class List {
    private:
        class ListNode;

    public:
        class ForwardIterator {
        public:
            using value_type = T;
            using reference = T &;
            using pointer = T *;
            using difference_type = ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;

            ForwardIterator(ListNode *node) : Ptr(node) {};

            T &operator*();
```

```
        ForwardIterator &operator++();

        ForwardIterator operator++(int);

        bool operator==(const ForwardIterator &it) const;

        bool operator!=(const ForwardIterator &it) const;

    private:
        ListNode *Ptr;

        friend class List;
    };

    ForwardIterator begin();

    ForwardIterator end();

    T& Top();

    void Insert(size_t & index,const T &value );

    void InsertHelp(const ForwardIterator &it, const T &value);

    void Erase(const ForwardIterator &it);

    bool Empty() {
        return Size == 0;
    }

    void Pop();

    void Push_back(const T &value);

    void Push(const T &value);

    List() = default;

    List(const List &) = delete;

    List &operator=(const List &) = delete;

    size_t Size = 0;
private:
    class ListNode {
```

```
        public:
            T Value;
            std::unique_ptr<ListNode> NextNode = nullptr;

            ForwardIterator next();

            ListNode(const T &value, std::unique_ptr<ListNode> next) : Value(value),
NextNode(std::move(next)) {};
        };

        std::unique_ptr<ListNode> Head = nullptr;


    };


    template<class T>
    typename List<T>::ForwardIterator List<T>::ListNode::next() {
        return {NextNode.get()};
    }

    template<class T>
    T &List<T>::ForwardIterator::operator*() {
        return Ptr->Value;
    }

    template<class T>
    typename List<T>::ForwardIterator &List<T>::ForwardIterator::operator++() {
        *this = Ptr->next();
        return *this;
    }

    template<class T>
    typename List<T>::ForwardIterator List<T>::ForwardIterator::operator++(int) {
        ForwardIterator prev = *this;
        ++(*this);
        return prev;
    }

    template<class T>
    bool List<T>::ForwardIterator::operator!=(const ForwardIterator &other) const {
        return Ptr != other.Ptr;
    }

    template<class T>
```

```
bool List<T>::ForwardIterator::operator==(const ForwardIterator &other) const {
   return Ptr == other.Ptr;
}

template<class T>
typename List<T>::ForwardIterator List<T>::begin() {
   return Head.get();
}

template<class T>
typename List<T>::ForwardIterator List<T>::end() {
   return nullptr;
}


template<class T>
void List<T>::Insert(size_t &index, const T &value) {
   int id = index - 1;
   if (index > this->Size) {
      throw std::logic_error("Out of bounds\n");
   }
   if (id == -1) {
      this->Push(value);
   }
   else {
      auto it = this->begin();
      for (int i = 0; i < id; ++i) {
         ++it;
      }
      this->InsertHelp(it,value);
   }
}


template<class T>
void List<T>::InsertHelp(const ForwardIterator &it, const T &value) {
   std::unique_ptr<ListNode> newNode(new ListNode(value, nullptr));
   if (it.Ptr == nullptr && Size != 0) {
      throw std::logic_error("Out of bounds");
   }
   if (Size == 0) {
      Head = std::move(newNode);
      ++Size;
   } else {
      newNode->NextNode = std::move(it.Ptr->NextNode);
```

```
        it.Ptr->NextNode = std::move(newNode);
        ++Size;
    }
}

template<class T>
void List<T>::Push(const T &value) {
    std::unique_ptr<ListNode> newNode(new ListNode(value, nullptr));
    newNode->NextNode = std::move(Head);
    Head = std::move(newNode);
    ++Size;
}

template<class T>
void List<T>::Push_back(const T &value) {
    size_t index =  Size;
    this->Insert(index,value);
}
template<class T>
T& List<T>::Top() {
    if (Head.get()) {
        return Head->Value;
    } else {
        throw std::logic_error("List is empty");
    }
}

template<class T>
void List<T>::Pop() {
    if (Head.get() == nullptr) {
        throw std::logic_error("List is empty");
    }
    Head = std::move(Head->NextNode);
    --Size;
}

template<class T>
void List<T>::Erase(const List<T>::ForwardIterator &it) {
    if (it.Ptr == nullptr) {
        throw std::logic_error("Invalid iterator");
    }
    --Size;
    if (it == this->begin()) {
        Head = std::move(Head->NextNode);
```

```
        } else {
            auto tmp = this->begin();
            while (tmp.Ptr->next() != it.Ptr) {
                ++tmp;
            }
            tmp.Ptr->NextNode = std::move(it.Ptr->NextNode);
        }
    }

}
```

Myallocator.h:

```
#pragma once

#include <cstdlib>
#include <cstdint>

#include <exception>
#include <iostream>
#include <type_traits>

#include "mylist.h"

namespace myal {

    template<class T, size_t ALLOC_SIZE>
    struct my_allocator {
        using value_type = T;
        using size_type = std::size_t;
        using difference_type = std::ptrdiff_t;
        using is_always_equal = std::false_type;

        template<class U>
        struct rebind {
            using other = my_allocator<U, ALLOC_SIZE>;
        };

        my_allocator() :
                memory_pool_begin_(new char[ALLOC_SIZE]),
                memory_pool_end_(memory_pool_begin_ + ALLOC_SIZE),
                memory_pool_tail_(memory_pool_begin_) {}

        my_allocator(const my_allocator &) = delete;

        my_allocator(my_allocator &&) = delete;
```

```
    ~my_allocator() {
        delete[] memory_pool_begin_;
    }

    T *allocate(std::size_t n);

    void deallocate(T *ptr, std::size_t n);

private:
    cntr::List<char*> free_blocks_;
    char *memory_pool_begin_;
    char *memory_pool_end_;
    char *memory_pool_tail_;


};

template<class T, size_t ALLOC_SIZE>
T *my_allocator<T, ALLOC_SIZE>::allocate(std::size_t n) {
    if (n != 1) {
        throw std::logic_error("This allocator can't allocate arrays");
    }
    if (size_t(memory_pool_end_ - memory_pool_tail_) < sizeof(T)) {
        if (!free_blocks_.Empty()) {
            auto it = free_blocks_.begin();
            char *ptr = *it;
            free_blocks_.Erase(it);
            return reinterpret_cast<T *>(ptr);
        }
        throw std::bad_alloc();
    }

    T *result = reinterpret_cast<T *>(memory_pool_tail_);
    memory_pool_tail_ += sizeof(T);
    return result;
}

template<class T, size_t ALLOC_SIZE>
void my_allocator<T, ALLOC_SIZE>::deallocate(T *ptr, std::size_t n) {
    if (n != 1) {
        throw std::logic_error("This allocator can't allocate arrays");
    }
    if (ptr == nullptr) {
        return;
    }
```

```
        free_blocks_.Push_back(reinterpret_cast<char*> (ptr));
    }

}
```

Mystack.h:

```
#pragma once

#include <iterator>
#include <memory>

namespace cntr {

    template<class T,class Allocator = std::allocator<T>>
    class Stack {
    private:
        class StackNode;
    public:
        class ForwardIterator {
        public:
            using value_type = T;
            using reference = T &;
            using pointer = T *;
            using difference_type = ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;

            ForwardIterator(StackNode *node) : Ptr(node) {};

            T &operator*();

            ForwardIterator &operator++();

            ForwardIterator operator++(int);

            bool operator==(const ForwardIterator &it) const;

            bool operator!=(const ForwardIterator &it) const;

        private:
            StackNode *Ptr;

            friend class Stack;
        };

        ForwardIterator begin();
```

```
    ForwardIterator end();

    T& Top();

    void Insert(int& index,const T &value );

    void InsertHelp(const ForwardIterator &it, const T &value);

    void Erase(const ForwardIterator &it);

    void Pop();

    void Push(const T &value);

    Stack() = default;

    Stack(const Stack &) = delete;

    Stack &operator=(const Stack &) = delete;

    size_t Size = 0;
private:
    using allocator_type = typename Allocator::template rebind<StackNode>::other;

    struct deleter {
        deleter(allocator_type* allocator): allocator_(allocator) {}

        void operator() (StackNode* ptr) {
            if(ptr != nullptr){
                std::allocator_traits<allocator_type>::destroy(*allocator_, ptr);
                allocator_->deallocate(ptr, 1);
            }
        }

    private:
        allocator_type* allocator_;

    };

    using unique_ptr = std::unique_ptr<StackNode, deleter>;
    class StackNode {
    public:
        T Value;
        unique_ptr NextNode = {nullptr,deleter{&this->allocator_}};
```

```
        ForwardIterator next();

        StackNode(const T &value, unique_ptr next) : Value(value), NextNode(std::move(next)) {};
    };

    allocator_type allocator_{};
    unique_ptr Head = {nullptr,deleter{&this->allocator_}};


};


template<class T,class Allocator>
typename Stack<T,Allocator>::ForwardIterator Stack<T,Allocator>::StackNode::next() {
    return {NextNode.get()};
}

template<class T,class Allocator>
T &Stack<T,Allocator>::ForwardIterator::operator*() {
    return Ptr->Value;
}

template<class T,class Allocator>
typename Stack<T,Allocator>::ForwardIterator &Stack<T,Allocator>::ForwardIterator::operator++() {
    *this = Ptr->next();
    return *this;
}

template<class T,class Allocator>
typename Stack<T,Allocator>::ForwardIterator Stack<T,Allocator>::ForwardIterator::operator++(int) {
    ForwardIterator prev = *this;
    ++(*this);
    return prev;
}

template<class T,class Allocator>
bool Stack<T,Allocator>::ForwardIterator::operator!=(const ForwardIterator &other) const {
    return Ptr != other.Ptr;
}
```

```
template<class T,class Allocator>
bool Stack<T,Allocator>::ForwardIterator::operator==(const ForwardIterator &other) const {
    return Ptr == other.Ptr;
}

template<class T,class Allocator>
typename Stack<T,Allocator>::ForwardIterator Stack<T,Allocator>::begin() {
    return Head.get();
}

template<class T,class Allocator>
typename Stack<T,Allocator>::ForwardIterator Stack<T,Allocator>::end() {
    return nullptr;
}


template<class T,class Allocator>
void Stack<T,Allocator>::Insert(int &index, const T &value) {
    int id = index - 1;
    if (index < 0 || index > this->Size) {
        throw std::logic_error("Out of bounds\n");
    }
    if (id == -1) {
        this->Push(value);
    }
    else {
        auto it = this->begin();
        for (int i = 0; i < id; ++i) {
            ++it;
        }
        this->InsertHelp(it,value);
    }
}


template<class T,class Allocator>
void Stack<T,Allocator>::InsertHelp(const ForwardIterator &it, const T &value) {
    StackNode* newptr = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_,newptr,value,std::unique_ptr<StackNode,deleter>(
            nullptr,deleter{&this->allocator_}));
    unique_ptr newNode(newptr,deleter{&this->allocator_});
    if (it.Ptr == nullptr && Size != 0) {
```

```
            throw std::logic_error("Out of bounds");
        }
        if (Size == 0) {
            Head = std::move(newNode);
            ++Size;
        } else {
            newNode->NextNode = std::move(it.Ptr->NextNode);
            it.Ptr->NextNode = std::move(newNode);
            ++Size;
        }
    }

    template<class T,class Allocator>
    void Stack<T,Allocator>::Push(const T &value) {

        StackNode* newptr = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_,newptr,value,
std::unique_ptr<StackNode,deleter>(
                nullptr,deleter{&this->allocator_}));
        unique_ptr newNode(newptr,deleter{&this->allocator_});
        newNode->NextNode = std::move(Head);
        Head = std::move(newNode);
        ++Size;
    }

    template<class T,class Allocator>
    T& Stack<T,Allocator>::Top() {
        if (Head.get()) {
            return Head->Value;
        } else {
            throw std::logic_error("Stack is empty");
        }
    }

    template<class T,class Allocator>
    void Stack<T,Allocator>::Pop() {
        if (Head.get() == nullptr) {
            throw std::logic_error("Stack is empty");
        }
        Head = std::move(Head->NextNode);
        --Size;
    }

    template<class T,class Allocator>
    void Stack<T,Allocator>::Erase(const Stack<T,Allocator>::ForwardIterator &it) {
```

```
        if (it.Ptr == nullptr) {
            throw std::logic_error("Invalid iterator");
        }
        Size--;
        if (it == this->begin()) {
            Head = std::move(Head->NextNode);

        } else {
            auto tmp = this->begin();
            while (tmp.Ptr->next() != it.Ptr) {
                ++tmp;
            }
            tmp.Ptr->NextNode = std::move(it.Ptr->NextNode);
        }
    }

}
```

Main.cpp:

```
#include <iostream>
#include "mystack.h"
#include "pentagon.h"
#include <algorithm>
#include "myallocator.h"
#include <map>
void menu() {
    std::cout << "1 - add(1 - push, 2 - insert  by iterator(enter index new elem)\n"
                 "2 - delete(1 - pop, 2 - delete by iterator(enter index)\n"
                 "3 - top\n"
                 "4 - print\n"
                 "5 - count if(enter max area)\n"
                 "6 - exit\n";
}

void usingStack() {
    int command, minicommand,index;
    double val;
    cntr::Stack<pentagon<double>,myal::my_allocator<pentagon<double>,330>> st;
    for (;;) {
        std::cin >> command;
          if (command == 1) {
            try {
          std::cin >> minicommand;
            if (minicommand == 1) {
                pentagon<double> p(std::cin);
                st.Push(p);
```

```
    } else if (minicommand == 2) {
        std::cin >> index;
        try {
            pentagon<double> p(std::cin);
            st.Insert(index,p);

        } catch (std::logic_error &e) {
            std::cout << e.what() << std::endl;
        continue;
        }
    } }
catch(std::bad_alloc& e) {
    std::cout << e.what() << std::endl;
    std::cout << "memory limit\n";
    continue; }
  } else if (command == 6) {
    break;
  } else if (command == 2) {
    std::cin >> minicommand;
    if (minicommand == 1) {
        try {
            st.Pop();
        } catch (std::logic_error &e) {
            std::cout << e.what() << std::endl;
            continue;
        }
    }
    if (minicommand == 2) {
        std::cin >> index;
        try {
            if (index < 0 || index > st.Size) {
                throw std::logic_error("Out of bounds\n");
            }
            auto it = st.begin();
            for (int i = 0; i < index; ++i) {
                ++it;
            }
            st.Erase(it);
        }
        catch (std::logic_error &e) {
            std::cout << e.what() << std::endl;
            continue;
        }
    }
  } else if (command == 3) {
```

```
            try {
                st.Top().print(std::cout);
            }
            catch (std::logic_error &e) {
                std::cout << e.what() << std::endl;
                continue;
            }
        } else if (command == 4) {
            for (auto elem: st) {
                elem.print(std::cout);
            }
        } else if (command == 5) {
            std::cin >> val;
            std::cout << std::count_if(st.begin(), st.end(), [val](pentagon<double> r) {
return r.area() < val; })
                      << std::endl;
        } else {
            std::cout << "Error command\n";
            continue;
        }
    }
}

int main() {
    menu();
    usingStack();
    std::map<int, int, std::less<int>,
    myal::my_allocator<std::pair<const int, int>, 80>> mp;
    for(int i = 0; i < 2; ++i){
        mp[i] = i;
    }
    for(int i = 2; i < 10; ++i){
        mp.erase(i - 2);
        mp[i] = i + 3;
    }
    return 0;
}
```

2. Ссылка на репозиторий на GitHub.

https://github.com/magomed2000kasimov/oop_exercise_06

3. Набор тестов.

test_01.test:

1 1

0 0
0 3
2 3
4 3
4 0

1 2 3

1 1 1 1 1 1 1 1 1 1

1 2 0

0 0
0 2
2 2
3 1
2 0

3
4

1 2 0

0 0
0 2
2 2
2 0
1 0

4
5 4.3
6

test_02.test:

1 1
0 0 0 0 0 0 0 0 0 0
1 2 0
1 1 1 1 1 1 1 1 1 1
1 2 2
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
4
1 1
2 2 2 2 2 2 2 2 2 2

test_03.test:

1 2 0
1 1 1 1 1 1 1 1 1 1

1 2 1
2 2 2 2 2 2 2 2 2 2
1 2 2
3 3 3 3 3 3 3 3 3 3
4
2 1
3
1 1
0 0 0 0 0 0 0 0 0 0
2 2 0
2 2 0
2 2 0
6

4. Результаты выполнения тестов.

test_01.result:

1 - add(1 - push, 2 - insert by iterator(enter index new elem)
2 - delete(1 - pop, 2 - delete by iterator(enter index)
3 - top
4 - print
5 - count if(enter max area)
6 - exit
Out of bounds

coordinate:
{
0 0
0 2
2 2
3 1
2 0
}
coordinate:
{
0 0
0 2
2 2
3 1
2 0
}
coordinate:
{
0 0
0 3
2 3

4 3
4 0
}
coordinate:
{
0 0
0 2
2 2
2 0
1 0
}
coordinate:
{
0 0
0 2
2 2
3 1
2 0
}
coordinate:
{
0 0
0 3
2 3
4 3
4 0
}
1
test_02.result:
1 - add(1 - push, 2 - insert by iterator(enter index new elem)
2 - delete(1 - pop, 2 - delete by iterator(enter index)
3 - top
4 - print
5 - count if(enter max area)
6 - exit
coordinate:
{
1 1
1 1
1 1
1 1
1 1
}
coordinate:
{

0 0
0 0
0 0
0 0
0 0
}
coordinate:
{
-1 -1
-1 -1
-1 -1
-1 -1
-1 -1
}
test_03.result:
1 - add(1 - push, 2 - insert by iterator(enter index new elem)
2 - delete(1 - pop, 2 - delete by iterator(enter index)
3 - top
4 - print
5 - count if(enter max area)
6 - exit
coordinate:
{
1 1
1 1
1 1
1 1
1 1
}
coordinate:
{
2 2
2 2
2 2
2 2
2 2
}
coordinate:
{
3 3
3 3
3 3
3 3
3 3
}

coordinate:
{
2 2
2 2
2 2
2 2
2 2
}

5. Объяснение результатов работы программы.

Площадь для пятиугольника находится методом Гаусса (метод шунтирования). В 1 тесте я выходил за границу и пытался удалить объект, которого в стеке нет. Как в результирующем файле все ошибки корректно обрабатываются благодаря конструкции try catch. Все как в прошлой лабораторной за исключением нового способа выделения и удаления памяти, а именно с помощью аллокатора.

6. Вывод.

Выполняя данную лабораторную, я получил опыт работы с аллокаторами и умными указателями в C++, с системой контроля версий git. Узнал о применении аллокаторов в STL, также узнал что умным указателям можно передавать функтор, с помощью которого они будут освобождать память. Понял, для чего вообще нужны аллокаторы. Замечательный C++ дает возможность программисту самому решать, как он хочет работать с памятью, а аллокаторы в этом помогают.

7.Литература.

1)лекции по ООП МАИ.
2) Г.Шилдт «C++».
3)Б.Страуструп «C++ специальное издание».