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: Аллокаторы

Студент: Пермяков Никита Александрович
Группа: 80-208
Преподаватель: Чернышов Л.Н.
Дата:
Оценка:

Москва, 2020

1. Постановка задачи

Цель:

Изучение основ работы с контейнерами, знакомство концепцией аллокаторов памяти.

Порядок выполнения работы:

1. Ознакомиться с теоретическим материалом.
2. Получить у преподавателя вариант задания.
3. Реализовать задание своего варианта в соответствии с поставленными требованиями.
4. Подготовить тестовые наборы данных.
5. Создать репозиторий на GitHub.
6. Отправить файлы лабораторной работы в репозиторий.
7. Отчитаться по выполненной работе путём демонстрации работающей программы на тестовых наборах данных (как подготовленных самостоятельно, так и предложенных преподавателем) и ответов на вопросы преподавателя (как из числа контрольных, так и по реализации программы).

Вариант: 5

реализовать:

фигура - Ромб
контейнер - Стек
аллокатор - Динамический массив

Репозиторий содержит файлы:

- main.cpp //файл с заданием работы
- CMakeLists.txt // файл с конфигураций CMake
- report.doc // отчет о лабораторной работе

Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения, т.е. равносторонние (кроме трапеции и прямоугольника). Для хранения координат фигур необходимо

использовать шаблон std::pair.
Например:

```
template <class T>
struct Rhombus{

    private:
        std::array<vertex_t<T>, 4> points;};
```

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;
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.
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;

2. Описание программы

Программа состоит из 5 файлов:

1) main.cpp
2) stack.h - реализация стека для хранения фигур
3) dynamic.h - реализация динамического массива для хранения пустых блоков памяти
4) allocator.h - реализация аллокатора и его основных методов
5) rhombus.h - реализация ромба и его методов

Аллокатор имеет 2 шаблонных параметра: тип хранимых данных и размер памяти в байтах. Аллокатор хранит динамический массив свободных блоков. В случае, когда часть занятой памяти освобождают, то ссылка на данный блок памяти хранится в динамическом массиве. Фигура добавляется в самый правый незаполненный участок памяти, то есть после хвоста. Если свободного участка памяти после хвоста и до конца участка памяти будет недостаточно, чтобы сохранить фигуру, то фигура будет сохранена в свободный блок памяти, если он есть в стеке. Иначе программа сообщит, что места недостаточно.

3. Набор тестов

Пояснение:

В двух тестах зададим разный размер памяти при создании стека, а также может добавить фигуру в стек [1], в верх стека [2] или после определенного элемента по индексу с помощью итератора [2], выводить все фигуры, содержащиеся в стеке[3]. Удалять фигуру в стеке [2]: верхнюю фигуру [4], либо удалять по указанному индексу с помощью итератора [2]. Также пользователь может с вызвать функцию [5] и задать площадь, после чего будет указано количество фигур, с площадью меньше заданного числа. Вызов меню [6]. Выход и программы [0]. В случае если памяти для добавления фигуры будет недостаточно, выйдет исключение.

В коде программы зададим размер стека 128 байт, которых хватит для хранения 3 фигур
“Stack<Rhombus<double>, allocator< Rhombus<double>, 128>> st;”

1) На ввод подается число выбора действия из текстового интерфейса
2) При обновлении координат фигуры – аллокатор проверяет наполненность, и в случае свободных блоков – сохраняет фигуру. В противном случае – удваивает выделенную память и записывает координаты.
3) Выводится информация о состоянии стека фигур, происходит проверка тестами
4) В случае если памяти для добавления фигуры будет недостаточно, выйдет исключение.
5) Также пользователь может с вызвать функцию count_if и задать площадь, после чего будет указано количество фигур, с площадью меньше заданного числа.
6) Удалять фигуру в стеке:
   - верхнюю фигуру;
   - удалять по указанному индексу с помощью итератора.
7) Вызов меню.
8) Выход и программы.

Test 1

1

1

1 4 4 5 3 2 0 1

4

3

5

4

5

34

6

0

Test 2

1

1

1 3 4 4 3 1 0 0

6

2

1

3

0

Test 3

1

1

1 3 4 4 3 1 0 0

6

2

1

3

1

1

1 4 4 5 3 2 0 1

3

1

2

2

1 4 4 5 3 2 0 1

3

1. Результаты выполнения тестов

Test 1

```
Enter number for action:
1 Add Rhombus to stack
2 Remove item
3 Show all Rhombuses with std::for_each
4 Show top stack element
5 Display on screen number of objects whose area is
       less than specified with std::count_if
6 Print Menu
0 Exit
1
Add item to top of stack - 1
        to iterator position - 2
1
push --> Input points: 1 4 4 5 3 2 0 1
4
top print --> Rhombus: [1.000, 4.000] [4.000, 5.000] [3.000, 2.000] [0.000, 1.000]
3
print -->
0       Rhombus: [1.000, 4.000] [4.000, 5.000] [3.000, 2.000] [0.000, 1.000]
       Rhombus area:   8.000
5
<count_if> --> Enter area:
4
The number of figures with area less than entered:      0


5
```

<count_if> --> Enter area:

34

The number of figures with area less than entered: 1

6

Enter number for action:

1 Add Rhombus to stack

2 Remove item

3 Show all Rhombuses with std::for_each

4 Show top stack element

5 Display on screen number of objects whose area is

    less than specified with std::count_if

6 Print Menu

0 Exit

0

Test 2

Enter number for action:

1 Add Rhombus to stack

2 Remove item

3 Show all Rhombuses with std::for_each

4 Show top stack element

5 Display on screen number of objects whose area is

    less than specified with std::count_if

6 Print Menu

0 Exit

1

Add item to top of stack - 1

    to iterator position - 2

1

```
push --> Input points: 1 4 4 5 3 2 0 1
4
top print --> Rhombus: [1.000, 3.000] [4.000, 4.000] [3.000, 1.000] [0.000, 0.000]
3
print -->
0       Rhombus: [1.000, 3.000] [4.000, 4.000] [3.000, 1.000] [0.000, 0.000]
        Rhombus area:   8.000
5
<count_if> --> Enter area:
2
The number of figures with area less than entered:      0


5
<count_if> --> Enter area:
37
The number of figures with area less than entered:      1
6
Enter number for action:
1 Add Rhombus to stack
2 Remove item
3 Show all Rhombuses with std::for_each
4 Show top stack element
5 Display on screen number of objects whose area is
        less than specified with std::count_if
6 Print Menu
0 Exit
0
```

Test 3

```
Enter number for action:
1 Add Rhombus to stack
2 Remove item
3 Show all Rhombuses with std::for_each
4 Show top stack element
5 Display on screen number of objects whose area is
       less than specified with std::count_if
6 Print Menu
0 Exit
1
Add item to top of stack - 1
        to iterator position - 2
1
push --> Input points: 1 4 4 5 3 2 0 1
4
top print --> Rhombus: [10.000, 4.000] [14.000, 5.000] [13.000, 2.000] [10.000, 1.000]
3
print -->
0       Rhombus: [10.000, 4.000] [14.000, 5.000] [13.000, 2.000] [10.000, 1.000]
       Rhombus area:   8.000
5
<count_if> --> Enter area:
6
The number of figures with area less than entered:      0


5
<count_if> --> Enter area:
```

```
23

The number of figures with area less than entered:      1

6

Enter number for action:

1 Add Rhombus to stack

2 Remove item

3 Show all Rhombuses with std::for_each

4 Show top stack element

5 Display on screen number of objects whose area is

        less than specified with std::count_if

6 Print Menu

0 Exit

0
```

5. Листинг программы

main.cpp

```
#include <algorithm>
#include <iostream>
#include "stack.h"
#include "rhombus.h"
#include "allocator.h"

void menu(){
    std::cout << std::endl << "Enter number for action:" << std::endl;
    std::cout << "1 Add Rhombus to stack" <<
    std::endl << "2 Remove item" <<
    std::endl << "3 Show all Rhombuses with std::for_each" <<
    std::endl << "4 Show top stack element" <<
    std::endl << "5 Display on screen number of objects whose area is\n\tless than specified
with std::count_if" <<
    std::endl << "6 Print Menu" <<
    std::endl << "0 Exit" << std::endl;
}

int main(){
    int index, threshold_area, cmd = 6;
```

```
Stack<Rhombus<double>, allocator<Rhombus<double>, 128>> st;
double area;
do {
    menu();
    std::cin >> cmd;
    switch(cmd) {
        case 1: {
            Rhombus<double> rhom;
            std::cout << "Add item to top of stack - 1\n\t to iterator position - 2" << std::endl;
            std::cin >> cmd;
            if(cmd == 1) {
                std::cout << "push --> Input points: ";
                try {
                    std::cin >> rhom;
                    st.push(rhom);
                } catch (std::exception &e) {
                    std::cout << e.what() << std::endl;
                    break;
                }
                continue;
            }
            else if (cmd == 2) {
                std::cout << "insert --> Input points: ";
                try {
                    std::cin >> rhom;
                } catch (std::exception &e) {
                    std::cout << e.what() << std::endl;
                    break;
                }
                std::cout << "       --> Input index: ";
                std::cin >> index;
                try {
                    auto it = st.begin();
                    for (size_t i = 0; i < index; ++i)
                        ++it;
                    st.insert(it, rhom);
                } catch (std::bad_alloc &e) {
                    std::cout << e.what() << std::endl;
                    break;
                }
                continue;
            }
            else {
                std::cout << "[Error 1] Invalid input" << std::endl;
                std::cin.clear();
                std::cin.ignore(30000, '\n');
                break;
            }
```

```
        }
        case 2: {
            std::cout << "Delete item from top on stack - 1\n\t to iterator position - 2" << std::en
dl;
            std::cin >> cmd;
            if (cmd == 1) {
                std::cout << "pop -->";
                try {
                    st.pop();
                } catch(std::logic_error &e){
                    std::cout << e.what() << std::endl;
                    break;
                }
                continue;
            }
            else if(cmd == 2) {
                std::cout << "erase --> Input index: ";
                std::cin >> index;
                try {
                    if(index < 0 || index > st.size)
                        throw std::logic_error("\nOut of bounds\n");
                    auto it = st.begin();
                    for(size_t i = 0; i < index; ++i)
                        ++it;
                    st.erase(it);
                } catch(std::logic_error &e) {
                    std::cout << e.what() << std::endl;
                    break;
                }
                continue;
            }
            else {
                std::cout << "[Error 2] Invalid input" << std::endl;
                std::cin.clear();
                std::cin.ignore(30000, '\n');
                break;
            }
        }
        case 3: {
            std::cout << "print -->" << std::endl;
            index = 0;
            std::for_each(st.begin(), st.end(), [&index](auto it){
                std::cout << index << "\t";
                it.print(std::cout);
                std::cout << "\tRhombus area:\t" << it.area() << std::endl;
                ++index;
            });
            continue;
```

```
            }
            case 4: {
                std::cout << "top print --> ";
                try{
                    auto it = st.top();
                    it.print(std::cout);
                } catch(std::logic_error &e){
                    std::cout << e.what() << std::endl;
                    break;
                }
                continue;
            }
            case 5: {
                std::cout << "<count_if> --> ";
                std::cout << "Enter area:\t" << std::endl;
                std::cin >> area;
                std::cout << "The number of figures with area less than entered:\t"
                << std::count_if(
                    st.begin(),
                    st.end(),
                    [area](Rhombus<double> thom){
                        return thom.area() < area;
                    }) << std::endl;
                continue;
            }
            case 6: {
                menu();
                continue;
            }
            case 0: {
                return 0;
            }
            default: {
                std::cout << "[Error 3] Invalid input" << std::endl;
                std::cin.clear();
                std::cin.ignore(30000, '\n');
            }
        }
    } while(true);
    return 0;
}
```

vertex.h

```
#pragma once

#include <iostream>

#include <cmath>
```

```
#include <iomanip>

template<typename T>
struct vertex_t {
    T x;
    T y;
};

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

template<typename T>
std::ostream& operator<< (std::ostream& os, const vertex_t<T>& p) {
    os  << std::fixed << std::setprecision(3) << "[" << p.x << ", " << p.y << "]";
    return os;
}

namespace vertex {
    template<typename T>
    T length(const vertex_t<T>& p1, const vertex_t<T>& p2) {
        return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2));
    }
}
```

stack.h

```
#pragma once
#include <memory>
#include <iterator>
#include <iostream>

template<typename T, typename Allocator=std::allocator<T>>
```

```
struct Stack {
private:
    struct Node;

public:
    Stack() = default;
    size_t size = 0;

    struct forward_iterator {
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = std::ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        forward_iterator(Node *ptr);
        T& operator* ();

        forward_iterator& operator++ ();
        forward_iterator operator++(int);

        bool operator== (const forward_iterator& o) const;
        bool operator!= (const forward_iterator& o) const;

    private:
        Node* ptr;
        friend struct Stack;
    };

    forward_iterator begin();
    forward_iterator end();

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

```
    void insert(const forward_iterator &it, const T& value);
    void erase(const forward_iterator &it);
    void push(const T &value);

private:
    using allocator_type = typename Allocator::template rebind<Node>::other;
    struct deleter {
        deleter(allocator_type* tmp_allocator): allocator(tmp_allocator) {}

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

    using unique_ptr = std::unique_ptr<Node, deleter>;
    unique_ptr head {nullptr, deleter{&this->allocator}};
    allocator_type allocator {};

    struct Node {
        Node() = default;
        Node(const T& value, unique_ptr next): value(value), next_node(std::move(next)) {};
        forward_iterator next();

        T value;
        unique_ptr next_node {nullptr, deleter{&this->allocator}};
    };
};
```

```
template<typename T, typename Allocator>
typename Stack<T, Allocator>::forward_iterator Stack<T, Allocator>::Node::next() {
   return this->next_node.get();
}

template<typename T, typename Allocator>
typename Stack<T, Allocator>::forward_iterator Stack<T, Allocator>::begin() {
   if (this->head == nullptr) {
      return nullptr;
   }
   return this->head.get();
}

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

template<typename T, typename Allocator>
Stack<T, Allocator>::forward_iterator::forward_iterator(Node *tmp_ptr): ptr(tmp_ptr) {}

template<typename T, typename Allocator>
T& Stack<T, Allocator>::forward_iterator::operator* () {
    return this->ptr->value;
}

template<typename T, typename Allocator>
typename Stack<T, Allocator>::forward_iterator &Stack<T, Allocator>::forward_iterator::opera
tor++ () {
    if(this->ptr != nullptr)
       *this = this->ptr->next();
   return *this;
}
```

```
template<typename T, typename Allocator>
typename Stack<T, Allocator>::forward_iterator Stack<T, Allocator>::forward_iterator::operator++ (int) {
    forward_iterator tmp = *this;
    ++(*this);
    return tmp;
}

template<typename T, typename Allocator>
bool Stack<T, Allocator>::forward_iterator::operator== (const forward_iterator &tmp) const {
   return this->ptr == tmp.ptr;
}

template<typename T, typename Allocator>
bool Stack<T, Allocator>::forward_iterator::operator!= (const forward_iterator &tmp) const {
   return this->ptr != tmp.ptr;
}

template<typename T, typename Allocator>
void Stack<T, Allocator>::insert(const forward_iterator &it, const T &value) {
    Node* new_ptr = this->allocator.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator, new_ptr, value, std::unique_ptr<Node, deleter>(nullptr, deleter{&this->allocator}));

    unique_ptr new_node(new_ptr, deleter{&this->allocator});

    if (it.ptr == nullptr && this->size != 0) {
        throw std::logic_error("Iter went beyonds of stack");
    } else if (it.ptr == nullptr && this->size == 0) {
        this->head = std::move(new_node);
        ++(this->size);
    } else {
        new_node->next_node = std::move(it.ptr->next_node);
        it.ptr->next_node = std::move(new_node);
```

```
        ++(this->size);
    }
}

template<typename T, typename Allocator>
void Stack<T, Allocator>::erase(const typename Stack<T, Allocator>::forward_iterator &it){
    if (it.ptr == nullptr) {
        throw std::logic_error("Erasing of iter is invalid");
    } else if (it == this->begin()) {
        this->head = std::move(it.ptr->next_node);
        --(this->size);
    } else {
        Stack<T, Allocator>::forward_iterator tmp_it = this->begin();
        while (tmp_it.ptr->next() != it.ptr)
            ++tmp_it;
        tmp_it.ptr->next_node = std::move(it.ptr->next_node);
    }
}

template<typename T, typename Allocator>
void Stack<T, Allocator>::push(const T &value) {
    Node* new_ptr = this->allocator.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator, new_ptr, value, std::unique
_ptr<Node, deleter>(nullptr, deleter{&this->allocator}));

    unique_ptr new_node(new_ptr, deleter{&this->allocator});
    new_node->next_node = std::move(this->head);
    this->head = std::move(new_node);
    ++(this->size);
}

template<typename T, typename Allocator>
T &Stack<T, Allocator>::top() {
    if (this->head.get())
```

```
        return this->head->value;
    throw std::logic_error("Stack is empty");
}

template<typename T, typename Allocator>
void Stack<T, Allocator>::pop() {
    if (this->head) {
        this->head = std::move(this->head->next_node);
        --(this->size);
   } else {
       throw std::logic_error("Stack is empty");
   }
}
```

thrombus.h

```
#pragma once
#include <array>
#include "vertex.h"

template<typename T>
struct Rhombus {
    Rhombus() {};
    Rhombus(const vertex_t<T>& p1, const vertex_t<T>& p2, const vertex_t<T>& p3, const v
ertex_t<T>& p4);

    T area() const;
    vertex_t<T> center() const;
    void print(std::ostream& os) const;

    void set_diagonal();
    void set_points(std::pair<T, T> &p, size_t &index);

    template<std::size_t N>
    T get_diagonal(const size_t (&arr_index)[N]);
```

```
    private:
        std::array<vertex_t<T>, 4> points;
        T small_diagonal, big_diagonal;
};

template<typename T>
template<std::size_t N>
T Rhombus<T>::get_diagonal(const size_t (&arr_index)[N]) {
    T d1 = vertex::length(this->points[arr_index[0]], this->points[arr_index[1]]);
    T d2 = vertex::length(this->points[arr_index[0]], this->points[arr_index[2]]);
    T d3 = vertex::length(this->points[arr_index[0]], this->points[arr_index[3]]);
    if(d1 == d2) {
        return d3;
    } else if(d1 == d3) {
        return d2;
    } else if(d2 == d3) {
        return d1;
    } else {
        throw std::invalid_argument("Entered coordinates are not forming Rhombus. Try enterin
g new coordinates");
    }
}

template<typename T>
void Rhombus<T>::set_diagonal(){
    try {
        T d1 = get_diagonal({0, 1, 2, 3});
        T d2 = get_diagonal({1, 0, 2, 3});
        T d3 = get_diagonal({2, 0, 1, 3});
        T d4 = get_diagonal({3, 0, 1, 2});
        if(d1 == d2 || d1 == d4) {
            if(d1 < d3) {
                this->small_diagonal = d1;
```

```
            this->big_diagonal = d3;
        } else {
            this->small_diagonal = d3;
            this->big_diagonal = d1;
        }
    } else if(d1 == d3) {
        if(d1 < d2) {
            this->small_diagonal = d1;
            this->big_diagonal = d2;
        } else {
            this->small_diagonal = d2;
            this->big_diagonal = d1;
        }
    }
  } catch(std::exception& e) {
    throw std::invalid_argument(e.what());
    return;
  }
}

template<typename T>
Rhombus<T>::Rhombus(const vertex_t<T>& p1, const vertex_t<T>& p2, const vertex_t<T>
& p3, const vertex_t<T>& p4) {
  this->points[0] = p1;
  this->points[1] = p2;
  this->points[2] = p3;
  this->points[3] = p4;
  this->set_diagonal();
}

template<typename T>
T Rhombus<T>::area() const {
  return this->small_diagonal * this->big_diagonal / 2.0;
}
```

```
template<typename T>
vertex_t<T> Rhombus<T>::center() const {
    if (vertex::length(this->points[0], this->points[1]) == this->small_diagonal ||
    vertex::length(this->points[0], this->points[1]) == this->big_diagonal) {
        return {
            ((this->points[0].x + this->points[1].x) / 2.0),
            ((this->points[0].y + this->points[1].y) / 2.0)
        };
    } else if (vertex::length(this->points[0], this->points[2]) == this->small_diagonal ||
    vertex::length(this->points[0], this->points[2]) == this->big_diagonal) {
        return {
            ((this->points[0].x + this->points[2].x) / 2.0),
            ((this->points[0].y + this->points[2].y) / 2.0)
        };
    } else {
        return {
            ((this->points[0].x + this->points[3].x) / 2.0),
            ((this->points[0].y + this->points[3].y) / 2.0)
        };
    }
}

template<typename T>
void Rhombus<T>::print(std::ostream& os) const {
    os << "Rhombus: ";
    for (const auto& p : this->points)
        os << p << ' ';
    os << std::endl;
}

template<typename T>
void Rhombus<T>::set_points(std::pair<T, T> &p, size_t &index){
    this->points[index].x = p.first;
```

```
    this->points[index].y = p.second;
}

template<typename T>
std::istream &operator>> (std::istream &is, Rhombus<T> &item) {
    std::pair<T, T> tmp_input;
    for (size_t i = 0; i < 4; ++i) {
        is >> tmp_input.first >> tmp_input.second;
        item.set_points(tmp_input, i);
    }
    item.set_diagonal();
    return is;
}
```

allocator.h

```
#pragma once
#include <exception>
#include <iostream>
#include "stack.h"

template<class T, size_t ALLOC_SIZE>
struct 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 = allocator<U, ALLOC_SIZE>;
   };

   allocator(): ptr_begin(new char[ALLOC_SIZE]), ptr_end(ptr_begin + ALLOC_SIZE), ptr_tail
(ptr_begin) {}
```

```
    allocator(const allocator&) = delete;
    allocator(allocator&&) = delete;

    ~allocator() {
        delete[] this->ptr_begin;
    }

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

private:
    char* ptr_begin;
    char* ptr_end;
    char* ptr_tail;
    Stack<char*> free_blocks;
};

template<class T, size_t ALLOC_SIZE>
T* allocator<T, ALLOC_SIZE>::allocate(std::size_t n){
     if(n != 1)
         throw std::logic_error("Can not allocate arrays");
     if(size_t(this->ptr_end - this->ptr_tail) < sizeof(T)){
         if(this->free_blocks.size == 0)
             throw std::bad_alloc();
         auto it = this->free_blocks.begin();
         char* ptr = *it;
         this->free_blocks.pop();
         return reinterpret_cast<T*>(ptr);
    }
    T* result = reinterpret_cast<T*>(this->ptr_tail);
    this->ptr_tail += sizeof(T);
    return result;
}
```

```
template<class T, size_t ALLOC_SIZE>
void allocator<T, ALLOC_SIZE>::deallocate(T* ptr, std::size_t n) {
  if(n != 1)
     throw std::logic_error("Can not allocate arrays");
  if(ptr == nullptr)
     return;
  this->free_blocks.push(reinterpret_cast<char*>(ptr));
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)

project(lab6)

add_executable(lab6 main.cpp)

set_property(TARGET lab6 PROPERTY CXX_STANDARD 11)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -g")
```

6. Ссылка на репозиторий

https://github.com/nikit34/oop_exercise_06

7. Объяснение результатов работы программы

Аллокатор повышает производительность, обычно использует оператор new для выделения памяти. Это реализуется как слой вокруг функций распределения кучи C, которые оптимизированы для нечастого выделения больших блоков памяти. Этот подход хорошо работает с контейнерами, которые в основном выделяют большие куски памяти, например vector и deque. Однако для контейнеров, которые требуют частого выделения небольших объектов, таких как мап или список, использование распределителя по умолчанию является медленнее. Другие распространенные проблемы с распределителем на основе malloc включают плохую локальность ссылок (?) и чрезмерную фрагментацию памяти. Популярным подходом к повышению производительности является создание распределителя памяти на основе пула. Вместо того, чтобы выделять память каждый раз, когда элемент вставляется или удаляется из контейнера, большой блок памяти (пул памяти) выделяется заранее, возможно, при запуске программы. Кастомный аллокатор будет обслуживать запросы выделения, возвращая указатель на память из пула. Фактическое освобождение памяти можно отложить до истечения срока службы пула памяти.

8. Вывод

В ходе работы научился реализовывать алокатор, с помощью которого можно избежать фрагментации памяти и ускорить работу программы за счет меньшего количества системных вызовов для выделения памяти. Алокатор основан на динамическом массиве.

Существуют следующие виды алокаторов:

1. линейный
2. основанный на объектном пуле
3. стековый
4. фреймовый
5. статический

9. Список литературы

1. Перегрузка операторов C++ [Электронный ресурс]. URL: https://metanit.com/cpp/tutorial/5.14.php

(дата обращения: 26.10.2020).

2. Академическое программирование C++ [Электронный ресурс]. URL:http://www.c-cpp.ru/books/akkadem.pdf

(дата обращения: 26.10.2020).