МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8 по курсу

объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Зубко Дмитрий Валерьевич, группа М80-208Б-20*
Преподаватель *Дорохов Евгений Павлович*

Цель работы:

Целью лабораторной работы является:

Закрепление навыков по работе с памятью в C++;
Создание аллокаторов памяти для динамических структур данных.

Задание:

Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных.
Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти. Аллокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианту задания). Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.

Нельзя использовать:

Стандартные контейнеры std.

Программа должна позволять:

Вводить произвольное количество фигур и добавлять их в контейнер;
Распечатывать содержимое контейнера;
Удалять фигуры из контейнера.

Дневник отладки

Во время выполнения лабораторной были некие трудности с реализацией линейного списка и аллокатора, позже они были полностью ликвидирован.

Недочёты

Недочётов не было обнаружено.

Выводы

Лабораторная работа №8 позволила мне реализовать свой класс

аллокаторов, полностью прочувствовать процесс выделения памяти на низкоуровневых языках программирования. Лабораторная прошла успешно. В процессе выполнения работы я на практике познакомился с понятием аллокатора. Так как во многих структурах данных используются аллокаторы, то это очень важная тема, которую должен знать каждый программист на C++. Написание собственноручного итератора помогает реализовать собственную логику выделения памяти, которая может быть более оправданной в некоторых ситуациях, чем стандартный аллокатор, как для самописных, так и для стандартных структур данных.

Исходный код

figure.h

```
#ifndef OOP5_FIGURE_H
#define OOP5_FIGURE_H

#include <cmath>
#include <iostream>
#include "point.h"

class Figure {
public:
    virtual size_t VertexesNumber() = 0;
    virtual double Area() = 0;
    virtual void Print(std::ostream &os) = 0;
    virtual ~Figure() {};
};
#endif //OOP5_FIGURE_H
```

main.cpp

```
#include "pentagon.h"
#include "TVector.h"
#include "TStack.h"

int main() {
    TVector<Pentagon> t;

    t.InsertLast(std::shared_ptr<Pentagon>(new Pentagon(
            {1.f, 9.f}, {8.f, 7.f}, {6.f, 5.f}, {4.f, 3.f}, {2.f, 1.f})));

    t.InsertLast(std::shared_ptr<Pentagon>(new Pentagon(
            {1.f, 9.f}, {8.f, 7.f}, {6.f, 5.f}, {4.f, 3.f}, {2.f, 1.f})));

    t.InsertLast(std::shared_ptr<Pentagon>(new Pentagon(
            {1.f, 9.f}, {8.f, 7.f}, {6.f, 5.f}, {4.f, 3.f}, {2.f, 1.f})));

    std::cout << t << std::endl;

    for (auto i = t.begin(); i != t.end(); ++i)
        (*i).Print(std::cout);
}
```

pentagon.cpp

```
#include "pentagon.h"

TAllocationBlock Pentagon::_alloc_block(sizeof(Pentagon), 32);

std::istream& operator>>(std::istream& is, Pentagon& p) {
    std::cout << "Enter data:" << std::endl;
    is >> p.a >> p.b >> p.c >> p.d >> p.e;
//    std::cout << "Pentagon created via istream" << std::endl;
    return is;
```

```
}

std::ostream& operator<<(std::ostream& os, Pentagon& p) {
    os << "Pentagon: " << p.a << p.b << p.c << p.d << p.e << std::endl;
    return os;
}


Pentagon& Pentagon::operator=(const Pentagon &other) {
    this->a = other.a;
    this->b = other.b;
    this->c = other.c;
    this->d = other.d;
    this->e = other.e;
    return *this;
}

bool Pentagon::operator==(const Pentagon &other) {
    return a == other.a && b == other.b && c == other.c && d == other.d && e ==
other.e;
}

size_t Pentagon::VertexesNumber() {
    return 5;
}

double Pentagon::SquareTriangle(Point a, Point b, Point c){
    double p = (a.dist(b) + b.dist(c) + c.dist(a)) / 2;
    return sqrt(p * (p - a.dist(b)) * (p - b.dist(c)) * (p - c.dist(a)));
}

double Pentagon::Area() {
    return SquareTriangle(a, b, c) + SquareTriangle(a, c, d) + SquareTriangle(a, d,
e);
}

void Pentagon::Print(std::ostream &os) {
    os << "Pentagon: " << a << b << c << d << e << std::endl;
}

Pentagon::Pentagon(){}

Pentagon::Pentagon(Point a_, Point b_, Point c_, Point d_, Point e_) : a(a_), b(b_),
c(c_), d(d_), e(e_) {}
```

```
Pentagon::Pentagon(const Pentagon &other) : Pentagon(other.a, other.b, other.c,
other.d, other.e) {
}

Pentagon::Pentagon(std::istream &is) {
    std::cout << "Enter data:" << std::endl;
    is >> a >> b >> c >> d >> e;
//    std::cout << "Pentagon created via istream" << std::endl;
}
```

Pentagon.h

```
#ifndef OOP5_PENTAGON_H
#define OOP5_PENTAGON_H

#include "figure.h"
#include "tallocation.h"

class Pentagon : Figure{
public:
    friend std::istream& operator>>(std::istream& is, Pentagon& p);
    friend std::ostream& operator<<(std::ostream& os, Pentagon& p);
    size_t VertexesNumber() override;
    double Area() override;
    void Print(std::ostream &os) override;
    bool operator==(const Pentagon& other);
    Pentagon();
    Pentagon(Point a_, Point b_, Point c_, Point d_, Point e_);
    Pentagon(std::istream &is);
    Pentagon(const Pentagon &other);
    Pentagon& operator=(const Pentagon& other);

    void *operator new(size_t size)
    {
        return _alloc_block.Allocate(size);
    }

    void operator delete(void *pointer)
    {
        _alloc_block.Free(pointer);
    }
private:
    Point a, b, c, d, e;
    double SquareTriangle(Point a, Point b, Point c);

    static TAllocationBlock _alloc_block;
};
```

```
#endif //OOP5_PENTAGON_H
```

Point.cpp

```
#include "point.h"

#include <cmath>

bool Point::operator==(const Point &other) {
    return (this->x_ == other.x_ && this->y_ == other.y_);
}

Point::Point() : x_(0.0), y_(0.0) {}

Point::Point(double x, double y) : x_(x), y_(y) {}

Point::Point(std::istream &is) {
    is >> x_ >> y_;
}

double Point::dist(Point& other) {
    double dx = (other.x_ - x_);
    double dy = (other.y_ - y_);
    return std::sqrt(dx*dx + dy*dy);
}

std::istream& operator>>(std::istream& is, Point& p) {
    is >> p.x_ >> p.y_;
    return is;
}

std::ostream& operator<<(std::ostream& os, Point& p) {
    os << "(" << p.x_ << ", " << p.y_ << ")";
    return os;
}
```

Point.h

```
#ifndef OOP5_POINT_H
#define OOP5_POINT_H
#include <iostream>

class Point {
```

```
public:
    Point();
    Point(std::istream &is);
    Point(double x, double y);

    double dist(Point& other);

    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);
    bool operator==(const Point& other);
private:
    double x_;
    double y_;
};

#endif //OOP5_POINT_H
```

TVector.cpp

```
#pragma once

#include <iostream>
#include <memory>
#include <cstdlib>

#include "TIterator.h"

template <typename T>
class TVector {
public:
    TVector();

    TVector(const TVector &);

    virtual ~TVector();

    size_t Length() const {
        return length_;
    }

    bool Empty() const {
        return !length_;
    }

    const std::shared_ptr<T> &operator[](const size_t index) const {
        return data_[index];
```

```
    }

    std::shared_ptr<T> &Last() const {
        return data_[length_ - 1];
    }

    void InsertLast(const std::shared_ptr<T> &);

    void EmplaceLast(const T &&);

    void Remove(const size_t index);

    T RemoveLast() {
        return *data_[--length_];
    }

    void Clear();

    TIterator<T> begin() {
        return TIterator<T>(data_);
    }

    TIterator<T> end() {
        return TIterator<T>(data_ + length_);
    }

    template<typename TF>
    friend std::ostream &operator<<(
            std::ostream &, const TVector<TF> &);

private:
    void _Resize(const size_t new_capacity);

    std::shared_ptr<T> *data_;
    size_t length_, capacity_;
};

template <typename T>
TVector<T>::TVector()
        : data_(new std::shared_ptr<T>[32]),
          length_(0), capacity_(32) {}

template <typename T>
TVector<T>::TVector(const TVector &vector)
        : data_(new std::shared_ptr<T>[vector.capacity_]),
          length_(vector.length_), capacity_(vector.capacity_) {
    std::copy(vector.data_, vector.data_ + vector.length_, data_);
}
```

```
template <typename T>
TVector<T>::~TVector() {
   delete[] data_;
}

template <typename T>
void TVector<T>::_Resize(const size_t new_capacity) {
   std::shared_ptr<T> *newdata = new std::shared_ptr<T>[new_capacity];
   std::copy(data_, data_ + capacity_, newdata);
   delete[] data_;
   data_ = newdata;
   capacity_ = new_capacity;
}

template <typename T>
void TVector<T>::InsertLast(const std::shared_ptr<T> &item) {
   if (length_ >= capacity_)
     _Resize(capacity_ << 1);
   data_[length_++] = item;
}

template <typename T>
void TVector<T>::EmplaceLast(const T &&item) {
   if (length_ >= capacity_)
      _Resize(capacity_ << 1);
   data_[length_++] = std::make_shared<T>(item);
}

template <typename T>
void TVector<T>::Remove(const size_t index) {
   std::copy(data_ + index + 1, data_ + length_, data_ + index);
   --length_;
}

template <typename T>
void TVector<T>::Clear() {
   delete[] data_;
   data_ = new std::shared_ptr<T>[32];
   length_ = 0;
   capacity_ = 32;
}

template <typename T>
std::ostream &operator<<(std::ostream &os, const TVector<T> &vector) {
   const size_t last = vector.length_ - 1;

   for (size_t i = 0; i < vector.length_; ++i)
      os << (*vector.data_[i]);
   os << std::endl;
```

```
    return os;
}
```

TIterator.h

```
#pragma once

#include <memory>

template <typename T>
class TIterator {
public:
    TIterator(std::shared_ptr<T> *iter) : iter_(iter) {}

    T operator*() const {
        return *(*iter_);
    }

    T operator->() const {
        return *(*iter_);
    }

    void operator++() {
        iter_ += 1;
    }

    TIterator operator++(int) {
        TIterator iter(*this);
        ++(*this);
        return iter;
    }

    bool operator==(TIterator const &iterator) const {
        return iter_ == iterator.iter_;
    }

    bool operator!=(TIterator const &iterator) const {
        return iter_ != iterator.iter_;
    }

private:
    std::shared_ptr<T> *iter_;
};
```

tallocation.h

```
#ifndef OOP6_TALLOCATION_H
#define OOP6_TALLOCATION_H

#include "tstack.h"

class TAllocationBlock {
public:
    TAllocationBlock(size_t size, size_t count);

    ~TAllocationBlock();

    void *Allocate(size_t size);

    void Free(void *pointer);

    inline bool FreeBlocksAvailable() const {
        return budget_;
    }

private:
    void _Resize(size_t new_count);

    size_t size_;
    size_t count_;
    size_t budget_;

    char *used_blocks_;
    TStack<void *> free_blocks_;
};
#endif //OOP6_TALLOCATION_H
```

tallocation.cpp

```
#include "tallocation.h"

#include <iostream>

TAllocationBlock::TAllocationBlock(size_t size, size_t count)
        : size_(size), count_(count), budget_(count),
          used_blocks_(new char[size * count])
{
```

```
    for (size_t i = 0; i < count; ++i)
        free_blocks_.Emplace((void *)(used_blocks_ + (i * size)));
}

TAllocationBlock::~TAllocationBlock()
{
    delete[] used_blocks_;
}

void TAllocationBlock::_Resize(size_t new_count)
{
    char *newdata = new char[size_ * new_count];
    std::copy(used_blocks_, used_blocks_ + (size_ * count_), newdata);
    delete[] used_blocks_;
    used_blocks_ = newdata;
    count_ = new_count;
}

void *TAllocationBlock::Allocate(size_t size)
{
    if (size != size_) {
        std::cerr << "This block allocates " << size_ << "bytes only. "
                  << "You tried to allocate " << size << '\n';
        return 0;
    }

    if (!budget_) {
        size_t old_cound = count_;
        _Resize(count_ << 1);
        budget_ += (count_ - old_cound);

        for (size_t i = old_cound; i < count_; ++i)
            free_blocks_.Emplace((void *)(used_blocks_ + (i * size_)));
    }

    --budget_;

    return free_blocks_.Pop();
}

void TAllocationBlock::Free(void *pointer)
{
    free_blocks_.Push(std::make_shared<void *>(pointer));
    ++budget_;
}
```

TStack.h

```
#pragma once

#include <ostream>
#include <memory>
#include <cstdlib>

template <typename T>
class TStack {
public:
    TStack();

    TStack(const TStack &);

    virtual ~TStack();

    size_t Length() const {
        return length_;
    }

    bool Empty() const {
        return !length_;
    }

    std::shared_ptr<T> &Top() const {
        return data_[length_ - 1];
    }

    void Emplace(const T &&);

    void Push(const std::shared_ptr<T> &);

    inline T Pop() {
        return *data_[--length_];
    }

    void Clear();

    template<typename TF>
    friend std::ostream &operator<<(
            std::ostream &, const TStack<TF> &);

private:
    void _Resize(const size_t new_capacity);

    std::shared_ptr<T> *data_;
```

```
    size_t length_, capacity_;
};

template <typename T>
TStack<T>::TStack()
        : data_(new std::shared_ptr<T>[32]),
          length_(0), capacity_(32) {}

template <typename T>
TStack<T>::TStack(const TStack &vector)
        : data_(new std::shared_ptr<T>[vector.capacity_]),
          length_(vector.length_), capacity_(vector.capacity_) {
    std::copy(vector.data_, vector.data_ + vector.length_, data_);
}

template <typename T>
TStack<T>::~TStack() {
    delete[] data_;
}

template <typename T>
void TStack<T>::_Resize(const size_t new_capacity) {
    std::shared_ptr<T> *newdata = new std::shared_ptr<T>[new_capacity];
    std::copy(data_, data_ + capacity_, newdata);
    delete[] data_;
    data_ = newdata;
    capacity_ = new_capacity;
}

template <typename T>
void TStack<T>::Emplace(const T &&item)
{
    if (length_ >= capacity_)
        _Resize(capacity_ << 1);
            data_[length_++] = std::make_shared<T>(item);
}


template <typename T>
void TStack<T>::Push(const std::shared_ptr<T> &item) {
    if (length_ >= capacity_)
      _Resize(capacity_ << 1);
    data_[length_++] = item;
}
```

```
template <typename T>
void TStack<T>::Clear() {
    delete[] data_;
    data_ = new std::shared_ptr<T>[32];
    length_ = 0;
    capacity_ = 32;
}

template <typename T>
std::ostream &operator<<(std::ostream &os, const TStack<T> &stack) {
    for (size_t i = stack.length_ - 1; i >= 0; --i)
        os << (*stack.data_[i]);
    return os;
}
```