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

ЛАБОРАТОРНАЯ РАБОТА №8

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Тихонов Фёдор Андреевич, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

Задание:

Используя структуру данных, разработанную для лабораторной работы №7, спроектировать и разработать аллокатор памяти для динамической структуры данных. Целью построения аллокатора является 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 операторы `new` и `delete` у классов-фигур.

Вариант №26:

- Фигуры: Квадрат, Прямоугольник, Трапеция
- Контейнер первого уровня: Очередь
- Контейнер второго уровня: Связный список

Описание программы:

Исходный код разделён на 17 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- rectangle.h – описание класса прямоугольника (наследуется от фигуры)
- rectangle.cpp – реализация класса прямоугольника
- TQueueItem.h – описание элемента очереди
- TQueueItem.cpp – реализация элемента очереди
- TQueueItem.h – описание очереди
- TQueueItem.cpp – реализация очереди
- TIterator.h – реализация итератора
- TAllocatorBlock.h/cpp – реализация класса аллокатора для фигуры
- TLinkedList.h/cpp – реализация класса связного списка для использования в аллокаторе
- TLinkedListItem.h/cpp – реализация класса элемента связного списка для использования в аллокаторе
- main.cpp – основная программа

Дневник отладки:

При выполнении работы ошибок выявлено не было.

Вывод:

В процессе выполнения работы я на практике познакомился с понятием аллокатора. Написание собственноручного аллокатора 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:

point.h:

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
    Point();
    Point(std::istream &is);
    Point(double x, double y);
    double fx();
    double fy();
    double dist(Point& other);
    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);

private:
    double x_;
    double y_;
};

#endif //POINT_H
```

point.cpp:

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

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::fx(){
    return x_;
};

double Point::fy(){
    return 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;
}
```

figure.h:

```
#ifndef FIGURE_H
#define FIGURE_H

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

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

#endif //FIGURE_H
```

rectangle.h:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include <iostream>
#include "point.h"
#include "figure.h"
#include "TAllocationBlock.h"

class Rectangle : public Figure {
public:
    Rectangle();
    Rectangle(Point a, Point b, Point c, Point d);
    Rectangle(std::istream& is);

    size_t VertexesNumber();
    double Area();
    void Print(std::ostream& os);

    void * operator new (size_t size);
    void operator delete(void *ptr);

private:
    Point a_;
```

```
    Point b_;
    Point c_;
    Point d_;
    static TAllocationBlock block;
};

#endif //RECTANGLE_H
```

rectangle.cpp:

```
#include <iostream>
#include "point.h"
#include "rectangle.h"

Rectangle::Rectangle() : a_(Point()), b_(Point()), c_(Point()), d_(Point()) {}

Rectangle::Rectangle(Point a, Point b, Point c, Point d) : a_(a), b_(b), c_(c), d_(d) {}

Rectangle::Rectangle(std::istream& is) {
    is >> a_ >> b_ >> c_ >> d_;
}

void Rectangle::Print(std::ostream& os) {
    os << "Rectangle: " << a_ << " " << b_ << " " << c_ << " " << d_ << std::endl;
}

size_t Rectangle::VertexesNumber() {
    return 4;
}

double Rectangle::Area(){
    return a_.dist(b_) * c_.dist(d_);
}

TAllocationBlock Rectangle::block(sizeof(Rectangle), 1000);

void *Rectangle::operator new(size_t size) {
    return block.Allocate();
}

void Rectangle::operator delete(void *ptr) {
    block.Deallocate(ptr);
}
```

TQueueItem.h:

```
#ifndef FIGURE_H_TQUEUEITEM_H
#define FIGURE_H_TQUEUEITEM_H

#include "square.h"
#include "trapezoid.h"
#include "rectangle.h"
#include <memory>

template <class T> class TQueueItem {
public:
    TQueueItem(const std::shared_ptr<T> &poly);
    TQueueItem(const std::shared_ptr<TQueueItem<T>> &other);
```

```
    ~TQueueItem();

    template<class A> friend std::ostream& operator<<(std::ostream& os, const
std::shared_ptr<TQueueItem<A>> &poly);

public:
    std::shared_ptr<T> polygon;
    std::shared_ptr<TQueueItem<T>> next;
};

#define TQUEUEITEM_FUNCTIONS
#include "TQueueItem.cpp"

#endif //FIGURE_H_TQUEUEITEM_H
```

TQueueItem.cpp:

```
#ifndef TQUEUEITEM_FUNCTIONS
#include "TQueueItem.h"

#else

template <class T>
TQueueItem<T>::TQueueItem(const std::shared_ptr<T> &poly) {
    this->polygon = poly;
    this->next = nullptr;
}

template <class T>
TQueueItem<T>::TQueueItem(const std::shared_ptr<TQueueItem<T>> &other) {
    this->polygon = other->polygon;
    this->next = other->next;
}

template <class A>
std::ostream& operator<<(std::ostream& os, const std::shared_ptr<TQueueItem<A>> &poly) {
    os << "(" << poly->polygon << ")" << std::endl;
    return os;
}

template <class T>
TQueueItem<T>::~TQueueItem() = default;

#endif
```

TQueue.h:

```
#ifndef FIGURE_H_TQUEUE_H
#define FIGURE_H_TQUEUE_H

#include "TQueueItem.h"
#include "TIterator.h"
#include <iostream>

template <class T>
class TQueue {
public:
    TQueue();
    TQueue(const TQueue<T>& other);
```

```
    void Push(const std::shared_ptr<T> &&polygon);
    void Pop();
    std::shared_ptr<T> Top();
    bool Empty();
    size_t Length();

    TIterator<TQueueItem<T>, T> begin();
    TIterator<TQueueItem<T>, T> end();

    template<class A>
    friend std::ostream& operator<<(std::ostream& os, const TQueue<A>& queue); // "=> Sn Sn-1
... S1 =>"

    void Clear();
    ~TQueue();

private:
    size_t len;
    std::shared_ptr<TQueueItem<T>> head;
    std::shared_ptr<TQueueItem<T>> tail;
};



#define TQUEUE_FUNCTIONS
#include "TQueue.cpp"

#endif //FIGURE_H_TQUEUE_H
```

TQueue.cpp:

```
#ifndef TQUEUE_FUNCTIONS
#include "TQueue.h"

#else

template <class T>
TQueue<T>::TQueue() : head(nullptr), tail(nullptr), len(0) { }

template <class T>
TQueue<T>::TQueue(const TQueue<T>& other) {
    head = other.head;
    tail = other.tail;
    len = other.len;
}

template <class T>
void TQueue<T>::Push(const std::shared_ptr<T> &&polygon) {
    std::shared_ptr<TQueueItem<T>> new_tail =
            std::make_shared<TQueueItem<T>>(TQueueItem<T>(polygon));
    if (head != nullptr)
        tail->next = new_tail, tail = new_tail;
    else if (len == 1)
        head->next = new_tail, tail = new_tail;
    else
        head = tail = new_tail;
    len++;
}
```

```
template <class T>
void TQueue<T>::Pop() {
    if (len)
        head = head->next, len--;
}

template <class T>
std::shared_ptr<T> TQueue<T>::Top() {
    if (len)
        return head->polygon;
}

template <class T>
bool TQueue<T>::Empty() {
    return (len == 0);
}

template <class T>
size_t TQueue<T>::Length() {
    return len;
}

template <class T>
TIterator<TQueueItem<T>, T> TQueue<T>::begin() {
    return TIterator<TQueueItem<T>, T>(head);
}

template <class T> TIterator<TQueueItem<T>, T> TQueue<T>::end() {
    return TIterator<TQueueItem<T>, T>(nullptr);
}

template <class T>
std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {
    std::shared_ptr<TQueueItem<T>> item = queue.head;
    double sq[queue.len];
    for (int i = 0; i < (int)queue.len; i++) {
        sq[i] = item->polygon->Area();
        item = item->next;
    }
    os.precision(5);
    os << "=> ";
    for (int i = (int)queue.len - 1; i >= 0; i--) {
        os << sq[i] << " ";
    }
    os << "=>";
    return os;
}

template <class T>
void TQueue<T>::Clear() {
    std::shared_ptr<TQueueItem<T>> elem = head;
    std::shared_ptr<TQueueItem<T>> fore = head;
    while (elem) {
        fore.reset();
        fore = elem;
        elem = elem->next;
    }
    len = 0;
}
```

```
template <class T>
TQueue<T>::~TQueue() { }

#endif
```

TIterator.cpp:

```
#ifndef LAB5_TITERATOR_H
#define LAB5_TITERATOR_H

#include "square.h"

#include <iostream>

template <class item, class T> class TIterator {
public:
    TIterator(std::shared_ptr<item> n) {
        item_ptr = n;
    }

    std::shared_ptr<T> operator *() {
        return item_ptr->polygon;
    }

    std::shared_ptr<T> operator ->() {
        return item_ptr->polygon;
    }

    void operator ++() {
        item_ptr = item_ptr->next;
    }

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

    bool operator ==(TIterator const& i) {
        return (item_ptr == i.item_ptr);
    }

    bool operator !=(TIterator const& i) {
        return (item_ptr != i.item_ptr);
    }

private:
    std::shared_ptr<item> item_ptr;
};


#endif //LAB5_TITERATOR_H
```

TAllocationBlock.h:

```
#ifndef LAB6_TALLOCATIONBLOCK_H
#define LAB6_TALLOCATIONBLOCK_H

#include <iostream>
#include <cstdlib>
```

```
#include "TLinkedList.h"

class TAllocationBlock {
public:
    TAllocationBlock(int32_t size, int32_t count);

    void *Allocate();
    void Deallocate(void *ptr);
    bool Empty();
    int32_t Size();

    virtual ~TAllocationBlock();

private:
    char *used_bl;
    TLinkedList free_bl;
};

#endif //LAB6_TALLOCATIONBLOCK_H
```

TAllocationBlock.cpp:

```
#include "TAllocationBlock.h"

TAllocationBlock::TAllocationBlock(int32_t size, int32_t count) {
    used_bl = (char *)malloc(size * count);
    for (int32_t i = 0; i < count; ++i) {
        void *ptr = (void *)malloc(sizeof(void *));
        ptr = used_bl + i * size;
        free_bl.InsertLast(ptr);
    }
}

void *TAllocationBlock::Allocate() {
    if (!free_bl.Empty()) {
        void *res = free_bl.GetBlock();
        int32_t first = 1;
        free_bl.Remove(first);
        std::cout << "Rectangle created" << std::endl;
        return res;
    } else {
        throw std::bad_alloc();
    }
}

void TAllocationBlock::Deallocate(void *ptr) {
    free_bl.InsertFirst(ptr);
}

bool TAllocationBlock::Empty() {
    return free_bl.Empty();
}

int32_t TAllocationBlock::Size() {
    return free_bl.Length();
}

TAllocationBlock::~TAllocationBlock() {
    while (!free_bl.Empty()) {
```

```
        int32_t first = 1;
        free_bl.Remove(first);
    }
    free(used_bl);
    std::cout << "Rectangle deleted" << std::endl;
}
```

TLinkedListItem.h:

```
#ifndef LAB6_TLINKEDLISTITEM_H
#define LAB6_TLINKEDLISTITEM_H

#include <memory>

class TLinkedListItem {
public:
    TLinkedListItem(void *link);

    TLinkedListItem* SetNext(TLinkedListItem* next);
    TLinkedListItem* GetNext();
    void* GetBlock();

    virtual ~TLinkedListItem();
private:
    void* link;
    TLinkedListItem* next;
};

#endif // LAB6_TLINKEDLISTITEM_H
```

TLinkedListItem.cpp:

```
#include "TLinkedListItem.h"
#include <iostream>

TLinkedListItem::TLinkedListItem(void* link) {
    this->link = link;
    this->next = nullptr;
}

TLinkedListItem* TLinkedListItem::SetNext(TLinkedListItem* next) {
    TLinkedListItem* old = this->next;
    this->next = next;
    return old;
}

TLinkedListItem* TLinkedListItem::GetNext() {
    return this->next;
}

void* TLinkedListItem::GetBlock() {
    return this->link;
}

TLinkedListItem::~TLinkedListItem() {
}
```

TLinkedList.cpp:

```
#ifndef LAB6_TLINKEDLIST_H
#define LAB6_TLINKEDLIST_H

#include "TLinkedListItem.h"
#include <memory>
#include <iostream>

class TLinkedList {
public:
    TLinkedList();
    void InsertFirst(void *link);
    void InsertLast(void *link);
    void Insert(int position, void *link);
    int Length();
    bool Empty();
    void Remove(int &position);
    void Clear();

    void* GetBlock();

    virtual ~TLinkedList();
private:
    TLinkedListItem* first;
};

#endif // LAB6_TLINKEDLIST_H
```

TLinkedList.cpp:

```
#include "TLinkedList.h"

TLinkedList::TLinkedList() {
    first = nullptr;
}

void TLinkedList::InsertFirst(void* link) {
    auto *other = new TLinkedListItem(link);
    other->SetNext(first);
    first = other;
}

void TLinkedList::Insert(int position, void *link) {
    TLinkedListItem *iter = this->first;
    auto *other = new TLinkedListItem(link);
    if (position == 1) {
        other->SetNext(iter);
        this->first = other;
    } else {
        if (position <= this->Length()) {
            for (int i = 1; i < position - 1; ++i)
                iter = iter->GetNext();
            other->SetNext(iter->GetNext());
            iter->SetNext(other);
        }
    }
}

void TLinkedList::InsertLast(void *link) {
    auto *other = new TLinkedListItem(link);
```

```
    TLinkedListItem *iter = this->first;
    if (first != nullptr) {
        while (iter->GetNext() != nullptr) {
            iter = iter->SetNext(iter->GetNext());
        }
        iter->SetNext(other);
        other->SetNext(nullptr);
    }
    else {
        first = other;
    }
}

int TLinkedList::Length() {
    int len = 0;
    TLinkedListItem* item = this->first;
    while (item != nullptr) {
        item = item->GetNext();
        len++;
    }
    return len;
}

bool TLinkedList::Empty() {
    return first == nullptr;
}

void TLinkedList::Remove(int &position) {
    TLinkedListItem *iter = this->first;
    if (position <= this->Length()) {
        if (position == 1) {
            this->first = iter->GetNext();
        } else {
            int i = 1;
            for (i = 1; i < position - 1; ++i) {
                iter = iter->GetNext();
            }
            iter->SetNext(iter->GetNext()->GetNext());
        }

    } else {
        std::cout << "error" << std::endl;
    }
}

void TLinkedList::Clear() {
    first = nullptr;
}

void * TLinkedList::GetBlock() {
    return this->first->GetBlock();
}

TLinkedList::~TLinkedList() {
    delete first;
}
```

main.cpp:

```
#include <iostream>
```

```
#include <memory>
#include "point.h"
#include "figure.h"
#include "rectangle.h"
#include "TQueue.h"

void menu() {
    using namespace std;
    cout << "Enter 0 to exit\n";
    cout << "Enter 1 to print length of queue\n";
    cout << "Enter 2 to clear the queue\n";
    cout << "Enter 3 to know if the queue is empty\n";
    cout << "Enter 4 to pop the first element from queue\n";
    cout << "Enter 5 to push new Rectangle to queue\n";
    cout << "Enter 6 to print queue\n";
}

int main() {
    TQueue<Figure> a;
    std::shared_ptr<Figure> ptr;
    int n = -1;
    menu();
    while (n != 0) {
        std::cin >> n;
        if (n == 1) {
            std::cout << "Length of queue is " << a.Length() << std::endl;
        }
        if (n == 2) {
            a.Clear();
            std::cout << "Cleared" << std::endl;
        }
        if (n == 3) {
            if (a.Empty())
                std::cout << "Queue is empty" << std::endl;
            else
                std::cout << "Queue is not empty" << std::endl;
        }
        if (n == 4) {
            a.Pop();
            std::cout << "Popped" << std::endl;
        }
        if (n == 5) {
            std::cout << "Please, enter coordinates of Rectangle" << std::endl;
            a.Push( std::make_shared<Rectangle>(Rectangle(std::cin)));
            std::cout << "Done" << std::endl;
        }
        if (n == 6) {
            std::cout << a << std::endl;
        }
        if (n == 7) {
            for (auto x : a) {
                x->Print(std::cout);
            }
        }
    }
    auto s1 = new Rectangle;
    delete s1;
    return 0;
}
```

Результат работы:

```
Enter 0 to exit
Enter 1 to print length of queue
Enter 2 to clear the queue
Enter 3 to know if the queue is empty
Enter 4 to pop the first element from queue
Enter 5 to push new Rectangle to queue
Enter 6 to print queue
0
Rectangle created
Rectangle deleted

Process finished with exit code 0
```