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