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

**ЛАБОРАТОРНАЯ РАБОТА №6**

по курсу объектно-ориентированное программирование 3 семестр, 2021/22 уч. Год

Студент Абросимов Алексей Дмитриевич, группа М8О-207Б-20

Преподаватель Дорохов Евгений Павлович

**Условие**

Задание:Вариант 3: прямоугольник, Вектор, Бинарное дерево.

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. Алокатор 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.

**Описание программы**

Весь исходный код лежит в 12 файликах:

1. main.cpp — основная программа

2. item.h — описание класса элемента динамического массива

3. item.cpp — описание методов элемента дин.массива

4. iter.h — описание класса итератора и его методов

5. rectangle.h — описание класса прямоугольника

6. rectangle.cpp — описание методов прямоугольника

7. tvector.h — описание класса дин.массива

8. tvector.cpp — описание методов дин.массива.

9. Tree,h — описание класса бинарного дерева и его методов

10. TreeNode.h — описание класса узла бин.дерева и его методов

11. tallocation\_block.cpp -описание методов класса аллокатора

12. tallocation\_block.h — описание класса класса аллокатора

**Дневник отладки**

Результат работы программы:

TAllocationBlock: Memory init

TAllocationBlock: Memory init

TAllocationBlock: Allocate 1 of 3

a1 pointer value:1

TAllocationBlock: Allocate 2 of 3

TAllocationBlock: Allocate 3 of 3

a2 pointer value:2

TAllocationBlock: Deallocate block

TAllocationBlock: Deallocate block

TAllocationBlock: Deallocate block

TAllocationBlock: Memory freed

TAllocationBlock: Allocate 1 of 10

TVector item: created

TAllocationBlock: Allocate 2 of 10

TVector item: created

Rectangle coords (1,1) (1,1) (1,1) (1,1)

Rectangle coords (1,1) (1,1) (1,1) (1,2)

Rectangle coords (1,1) (1,1) (1,1) (1,2)

Rectangle coords (1,1) (1,1) (1,1) (1,1)

TVector item: deleted

TVector item: deleted

Rectangle was deleted

TAllocationBlock: Deallocate block

Rectangle was deleted

TAllocationBlock: Deallocate block

TAllocationBlock: Memory freed

**Недочёты**

**Выводы**

Данная лабораторная работа позволила мне ознакомиться с такой важной вещью в программировании , как аллокатор. Собственно написанные аллокаторы позволяют оптимизировать выделение памяти, ускорить процесс нахождения свободных блоков, распределить нагрузку на все доступные блоки памяти.

Ссылка на гитхаб: https://github.com/yungalexxxey/oop\_labs/tree/main/lab6

**Исходный код**

main.cpp

#include <iostream>

#include "rectangle.h"

#include "tvector.h"

#include "Tree.h"

void TestTVector()

{

TVector<Rectangle> vec;

vec.push\_back(std::shared\_ptr<Rectangle>(new Rectangle(1, 1, 1, 1, 1, 1, 1, 1)));

vec.push\_back(std::shared\_ptr<Rectangle>(new Rectangle(1, 1, 1, 1, 1, 1, 1, 2)));

for (auto i : vec)

{

std::cout << \*i << std::endl;

}

while (!vec.empty())

{

std::cout << \*vec.pop\_back() << std::endl;

}

}

void TestAllocationBlock()

{

TAllocationBlock allocator(sizeof(int), 3);

int \*a1 = nullptr;

int \*a2 = nullptr;

int \*a3 = nullptr;

a1 = (int \*)allocator.allocate();

\*a1 = 1;

std::cout << "a1 pointer value:" << \*a1 << std::endl;

a2 = (int \*)allocator.allocate();

\*a2 = 2;

a3 = (int \*)allocator.allocate();

\*a3 = 3;

std::cout << "a2 pointer value:" << \*a2 << std::endl;

allocator.deallocate(a1);

allocator.deallocate(a2);

allocator.deallocate(a3);

}

int main()

{

TestAllocationBlock();

TestTVector();

return 0;

}

item.cpp

#include "item.h"

#include <iostream>

template <class T>

Item<T>::Item(const std::shared\_ptr<T>& item)

: item(item){

std::cout << "TVector item: created" << std::endl;

}

template <class T>

TAllocationBlock Item<T>::tvec\_alloc(sizeof(Item<T>), 10);

template <class T>

std::shared\_ptr<T> Item<T>::Get() const {

return this->item;

}

template <class T>

std::shared\_ptr<Item<T>> Item<T>::GetNext() {

return this->next;

}

template <class T>

Item<T>::~Item() {

std::cout << "TVector item: deleted" << std::endl;

}

template <class T>

void Item<T>::SetNext(std::shared\_ptr<Item<T>>& next) {

this->next=next;

}

template <class A>

std::ostream& operator<<(std::ostream& os, const Item<A>& obj) {

os << "Item: " << \*obj.item << std::endl;

return os;

}

template <class T>

void Item<T>::forget(){

next=nullptr;

}

template <class T>

void\* Item<T>::operator new(size\_t size) {

return tvec\_alloc.allocate();

}

template <class T>

void Item<T>::operator delete(void\* p) {

tvec\_alloc.deallocate(p);

}

#include "rectangle.h"

template class Item<Rectangle>;

template std::ostream& operator<<(std::ostream& os,

const Item<Rectangle>& obj);

item.h

#ifndef ITEM\_H

#define ITEM\_H

#include <memory>

#include "tallocation\_block.h"

template <class T>

class Item

{

public:

Item(const std::shared\_ptr<T> &triangle);

std::shared\_ptr<T> Get() const;

template <class A>

friend std::ostream &operator<<(std::ostream &os, const Item<A> &obj);

void SetNext(std::shared\_ptr<Item<T>> &next);

std::shared\_ptr<Item<T>> GetNext();

void forget();

void \*operator new(size\_t size);

void operator delete(void \*p);

virtual ~Item();

private:

std::shared\_ptr<T> item;

std::shared\_ptr<Item<T>> next;

static TAllocationBlock tvec\_alloc;

};

#endif // ITEM\_H

iter.h

#ifndef ITER\_H

#define ITER\_H

#include <iostream>

#include <memory>

template <class node, class T>

class Iter

{

public:

Iter(std::shared\_ptr<node> n) { node\_ptr = n; }

std::shared\_ptr<T> operator\*() { return node\_ptr->Get(); }

std::shared\_ptr<T> operator->() { return node\_ptr->Get(); }

void operator++() { node\_ptr = node\_ptr->GetNext(); }

Iter operator++(int)

{

Iter iter(\*this);

++(\*this);

return iter;

}

bool operator==(Iter const &i) { return node\_ptr == i.node\_ptr; }

bool operator!=(Iter const &i) { return !(\*this == i); }

private:

std::shared\_ptr<node> node\_ptr;

};

#endif // ITER\_H

rectangle.cpp

#include "rectangle.h"

#include <math.h>

Rectangle::Rectangle():x1(0),y1(0),x2(1),y2(1),x3(0),y3(0),x4(0),y4(0){

}

Rectangle::Rectangle(int x1,int x2,int x3,int x4,int y1,int y2,int y3,int y4){

this->x1=x1;

this->x2=x2;

this->x3=x3;

this->x4=x4;

this->y1=y1;

this->y2=y2;

this->y3=y3;

this->y4=y4;

}

Rectangle::~Rectangle(){

std::cout<<"Rectangle was deleted\n";

}

Rectangle::Rectangle(std::istream&is){

std::cout <<"set x1 and y1:";

is >> x1 >> y1;

std::cout <<"set x2 and y2:";

is >> x2 >> y2;

std::cout <<"set x3 and y3:";

is >> x3 >> y3;

std::cout <<"set x4 and y4:";

is >> x4 >> y4;

}

void Rectangle::Print(std::ostream&os){

os << "Rectangle " << "(" <<x1<<" "<<y1<<")"<< "(" <<x2<<" "<<y2<<")"<< "(" <<x3<<" "<<y3<<")"<< "(" <<x4<<" " <<y4<<")" <<std::endl;

}

size\_t Rectangle::VertexesNumber(){

return 4;

}

bool Rectangle::isit(){

double perp;

double perp2;

perp=(x4-x1)\*(x2-x1)+(y4-y1)\*(y2-y1);

perp2=(x3-x4)\*(x3-x2)+(y3-y4)\*(y3-y2);

if((perp+perp2)==0) return true;

else return false;

}

double Rectangle::Area(){

double r1 = sqrt((x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2));

double r2 = sqrt((x2 - x3) \* (x2 - x3) + (y2 - y3) \* (y2 - y3));

double r3 = sqrt((x1 - x3) \* (x1 - x3) + (y1 - y3) \* (y1 - y3));

double p=(r1+r2+r3)/2;

double s= 2\*sqrt((p \* (p - r1) \* (p - r2) \* (p - r3)));

return s;

}

std::ostream& operator<<(std::ostream &out, const Rectangle &rec){

out << "Rectangle coords " <<"("<< rec.x1 << "," << rec.y1 << ")"<< " " <<"("<< rec.x2 << "," << rec.y2 << ")"<< " "<< "("<< rec.x3 << "," << rec.y3 << ")"<< " " << "("<<rec.x4 << "," << rec.y4 << ")\n";

return out;

}

std::istream& operator>>(std::istream &in,Rectangle &rec){

in >> rec.x1;

in >> rec.y1;

in >> rec.x2;

in >> rec.y2;

in >> rec.x3;

in >> rec.y3;

in >> rec.x4;

in >> rec.y4;

return in;

}

rectangle.h

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <iostream>

class Rectangle

{

public:

Rectangle();

Rectangle(int x1, int x2, int x3, int x4, int y1, int y2, int y3, int y4);

Rectangle(std::istream &is);

bool isit();

void Print(std::ostream &os);

size\_t VertexesNumber();

double Area();

~Rectangle();

friend std::ostream &operator<<(std::ostream &out, const Rectangle &rec);

friend std::istream &operator>>(std::istream &in, Rectangle &rec);

private:

double x1;

double y1;

double x2;

double y2;

double x3;

double y3;

double x4;

double y4;

};

#endif // RECTANGLE\_H

tallocation\_block.cpp

#include "tallocation\_block.h"

#include <iostream>

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count)

: \_size(size), \_count(count)

{

\_used\_blocks = (char \*)malloc(\_size \* \_count);

for (size\_t i = 0; i < \_count; ++i)

{

\_free\_blocks.insert(\_used\_blocks + i \* \_size);

}

\_free\_count = \_count;

std::cout << "TAllocationBlock: Memory init" << std::endl;

}

void TAllocationBlock::print()

{

\_free\_blocks.print();

}

void \*TAllocationBlock::allocate()

{

void \*result = nullptr;

if (\_free\_count > 0)

{

result = \_free\_blocks.get();

\_free\_blocks.remove(result);

\_free\_count--;

std::cout << "TAllocationBlock: Allocate " << (\_count - \_free\_count);

std::cout << " of " << \_count << std::endl;

}

else

{

std::cout << "TAllocationBlock: No memory exception :-)" << std::endl;

}

return result;

}

void TAllocationBlock::deallocate(void \*pointer)

{

std::cout << "TAllocationBlock: Deallocate block " << std::endl;

\_free\_blocks.insert(pointer);

\_free\_count++;

}

bool TAllocationBlock::has\_free\_blocks()

{

return \_free\_count > 0;

}

TAllocationBlock::~TAllocationBlock()

{

if (\_free\_count < \_count)

{

std::cout << "TAllocationBlock: Memory leak?" << std::endl;

}

else

{

std::cout << "TAllocationBlock: Memory freed" << std::endl;

}

while (!\_free\_blocks.empty())

{

\_free\_blocks.remove(\_free\_blocks.get());

}

free(\_used\_blocks);

}

tallocation\_block.h

#ifndef TALLOCATION\_BLOCK\_H

#define TALLOCATION\_BLOCK\_H

#include <cstdlib>

#include "Tree.h"

class TAllocationBlock

{

public:

TAllocationBlock(size\_t size, size\_t count);

void \*allocate();

void deallocate(void \*pointer);

bool has\_free\_blocks();

void print();

virtual ~TAllocationBlock();

private:

size\_t \_size;

size\_t \_count;

char \*\_used\_blocks;

Tree<void \*> \_free\_blocks;

size\_t \_free\_count;

};

#endif // TALLOCATION\_BLOCK\_H

Tree.h

#ifndef TREE\_H

#define TREE\_H

#include "TreeNode.h"

#include <iostream>

template <typename T>

class Tree

{

template <typename Type>

friend Type max(const Type &, const Type &);

public:

Tree();

~Tree();

void insert(const T &);

void remove(const T &);

T get();

bool empty();

void print() const;

private:

TreeNode<T> \*\_root;

void insert\_helper(TreeNode<T> \*\*, const T &);

void remove\_helper(TreeNode<T> \*\*, const T &);

void delete\_helper(TreeNode<T> \*);

void print\_helper(TreeNode<T> \*, int) const;

};

template <typename T>

Tree<T>::Tree() : \_root(0)

{

}

template <typename T>

bool Tree<T>::empty()

{

if (\_root->get\_data() != nullptr)

return true;

return false;

}

template <typename T>

T Tree<T>::get()

{

return \_root->get\_data();

}

template <typename T>

Tree<T>::~Tree()

{

delete\_helper(\_root);

}

template <typename T>

void Tree<T>::delete\_helper(TreeNode<T> \*node)

{

if (node != 0)

{

delete\_helper(node->\_left);

delete\_helper(node->\_right);

delete node;

}

}

template <typename T>

void Tree<T>::insert(const T &data)

{

insert\_helper(&\_root, data);

}

template <typename T>

void Tree<T>::insert\_helper(TreeNode<T> \*\*node, const T &data)

{

if (\*node == 0)

\*node = new TreeNode<T>(data);

else

{

if ((\*node)->\_data > data)

insert\_helper(&((\*node)->\_left), data);

else

{

if ((\*node)->\_data < data)

insert\_helper(&((\*node)->\_right), data);

}

}

}

template <typename T>

void Tree<T>::print() const

{

print\_helper(\_root, 0);

}

template <typename T>

void Tree<T>::print\_helper(TreeNode<T> \*node, int spaces) const

{

while (node != 0)

{

print\_helper(node->\_right, spaces + 5);

for (int i = 1; i < spaces; ++i)

std::cout << ' ';

std::cout << node->\_data << std::endl;

node = node->\_left;

spaces += 5;

}

}

template <typename T>

void Tree<T>::remove(const T &data)

{

remove\_helper(&\_root, data);

}

template <typename T>

void Tree<T>::remove\_helper(TreeNode<T> \*\*node, const T &data)

{

if ((\*node)->\_data == data)

{

TreeNode<T> \*del\_node = \*node;

if ((\*node)->\_left == 0 && (\*node)->\_right == 0)

{

\*node = 0;

delete del\_node;

}

else

{

if ((\*node)->\_left == 0)

{

\*node = (\*node)->\_right;

delete del\_node;

}

else

{

if ((\*node)->\_right == 0)

{

\*node = (\*node)->\_left;

delete del\_node;

}

else

{

TreeNode<T> \*p = \*node;

TreeNode<T> \*i = (\*node)->\_left;

while (i->\_right != 0)

{

p = i;

i = i->\_right;

}

\*node = i;

p->\_right = i->\_left;

i->\_right = del\_node->\_right;

i->\_left = p;

delete del\_node;

}

}

}

}

else

{

if ((\*node)->\_data > data)

remove\_helper(&((\*node)->\_left), data);

else

{

if ((\*node)->\_data < data)

remove\_helper(&((\*node)->\_right), data);

}

}

}

template <typename Type>

Type max(const Type &left, const Type &right)

{

return left > right ? left : right;

}

#endif

TreeNode.h

#ifndef TREENODE\_H

#define TREENODE\_H

template <typename T>

class Tree;

template <typename T>

class TreeNode

{

friend class Tree<T>;

public:

TreeNode();

TreeNode(const T &);

T get\_data() const;

private:

T \_data;

TreeNode<T> \*\_left;

TreeNode<T> \*\_right;

};

template <typename T>

TreeNode<T>::TreeNode() : \_left(0), \_right(0)

{

}

template <typename T>

TreeNode<T>::TreeNode(const T &data) : \_data(data),\_left(0),\_right(0)

{

}

template <typename T>

T TreeNode<T>::get\_data() const

{

return \_data;

}

#endif

tvector.cpp

#include "tvector.h"

#include "rectangle.h"

template <class T>

TVector<T>::TVector() : length(0), count(0)

{

}

template <class T>

int TVector<T>::size()

{

return this->count;

}

template <class T>

bool TVector<T>::empty()

{

return count == 0;

}

template <class T>

void TVector<T>::push\_back(std::shared\_ptr<T> newfig)

{

std::shared\_ptr<Item<T>> other(new Item<T>(newfig));

if (count == length)

{

length++;

count++;

std::shared\_ptr<std::shared\_ptr<Item<T>>[]> narr(new std::shared\_ptr<Item<T>>[length]);

for (int i = 0; i < length - 1; i++)

narr[i] = arr[i];

narr[length - 1] = other;

if (count - 1)

{

arr[count - 2]->SetNext(narr[count - 1]);

}

//free(arr);

arr = narr;

}

else if (count < length)

{

arr[count] = other;

count++;

if (count - 1)

{

arr[count - 2]->SetNext(arr[count - 1]);

}

}

}

template <class T>

TVector<T>::~TVector()

{

}

template <class T>

std::shared\_ptr<T> TVector<T>::pop\_back()

{

std::shared\_ptr<T> result;

if (length > 1)

{

std::shared\_ptr<std::shared\_ptr<Item<T>>[]> narr(new std::shared\_ptr<Item<T>>[length - 1]);

for (int i = 0; i < count - 1; i++)

{

narr[i] = arr[i];

}

result = arr[count - 1]->Get();

count--;

length--;

arr = narr;

return result;

}

else

{

count--;

length--;

return arr[0]->Get();

}

}

template <class T>

void TVector<T>::resize(int newlength)

{

if (newlength == length)

return;

if (newlength > length)

{

std::shared\_ptr<std::shared\_ptr<Item<T>>[]> narr(new std::shared\_ptr<Item<T>>[length]);

for (int i = 0; i < length; i++)

narr[i] = arr[i];

arr = narr;

length = newlength;

}

else

{

std::shared\_ptr<std::shared\_ptr<Item<T>>[]> narr(new std::shared\_ptr<Item<T>>[length]);

for (int i = 0; i < newlength; i++)

narr[i] = arr[i];

arr = narr;

count = newlength;

}

}

template <class T>

void TVector<T>::clear()

{

resize(1);

pop\_back();

length = 0;

count = 0;

}

template <class T>

void TVector<T>::erase(int pos)

{

if (count == 0)

{

std::cout << "Container is empty" << std::endl;

return;

}

std::shared\_ptr<std::shared\_ptr<Item<T>>[]> narr(new std::shared\_ptr<Item<T>>[length]);

int current\_index = 0;

for (int i = 0; i < count; i++)

{

if (i != pos - 1)

{

narr[current\_index] = arr[i];

current\_index++;

}

}

count--;

length--;

arr = narr;

}

template <class T>

Iter<Item<T>, T> TVector<T>::begin()

{

return Iter<Item<T>, T>(arr[0]);

}

template <class T>

Iter<Item<T>, T> TVector<T>::end()

{

return Iter<Item<T>, T>(nullptr);

}

//перегрузка операций

template <class T>

std::shared\_ptr<Item<T>> TVector<T>::operator[](int i)

{

if (i >= 0 && i < this->length)

return this->arr[i];

}

template <class T>

std::ostream &operator<<(std::ostream &out, TVector<T> &cont)

{

for (int i = 0; i < cont.count; i++)

{

out << "figure #" << i + 1 << "coords is " << \*cont[i];

}

return out;

}

template class TVector<Rectangle>;

template std::ostream &operator<<(std::ostream &out, TVector<Rectangle> &cont);

tvector.h

#ifndef TVECTOR\_H

#define TVECTOR\_H

#include <memory>

#include "item.h"

#include "Iter.h"

template <class T>

class TVector

{

private:

int length;

int count;

std::shared\_ptr<std::shared\_ptr<Item<T>>[]> arr;

public:

TVector();

~TVector();

int size();

bool empty();

void resize(int nindex);

void push\_back(std::shared\_ptr<T> newrec);

void erase(int pos);

std::shared\_ptr<T> pop\_back();

void clear();

Iter<Item<T>, T> begin();

Iter<Item<T>, T> end();

std::shared\_ptr<Item<T>> operator[](int i);

template <class A>

friend std::ostream &operator<<(std::ostream &out, TVector<A> &cont);

};

#endif // TVECTOR\_H