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

ЛАБОРАТОРНАЯ РАБОТА №8

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

Студент Хашимов Амир Азизович, группа М8О-207Б-20
Преподаватель Дорохов Евгений Павлович

Задание

Используя структуру данных, разработанную для лабораторной работы №7, 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.

Нельзя использовать:

- Стандартные контейнеры std.
- Программа должна позволять:
- Вводить произвольное количество фигур и добавлять их в контейнер;
- Распечатывать содержимое контейнера;
- Удалять фигуры из контейнера.

Вариант 27

GitHub [OOPLabs/lab6 at main · isAmirKhashimov/OOPLabs (github.com)]

Вывод

Данная лабораторная работа выдалась немного сложнее предыдущих, т.к. мне довелось познакомиться с работой аллокаторов, принципы которых я ранее не особо изучал. В целом, я могу сказать, что написанные программистом аллокаторы позволяют в некоторых случаях более грамотно выделять память. Однако при работе с ними требуется больше внимательности и осторожности, дабы избежать утечки память. Больших проблем не испытывал, но очень хорошо познакомился с работой аллокаторов.

Код программы:

```
// figure.cpp
#include <string>
#include "figure.h"


size_t Figure::VertexesNumber()
{
      return apixes.size();
}

// figure.h
```

```
#pragma once
#ifndef FIGURE_H_INCLUDED
#define FIGURE_H_INCLUDED

#include <ostream>
#include "point.h"
#include <vector>

class Figure
{
public:
	std::vector<Point> apixes;
	std::string figureName;
	virtual size_t VertexesNumber();
	virtual double Area() = 0;
	virtual void Print(std::ostream& os) = 0;
	virtual void Read(std::istream& is) = 0;

	friend std::ostream& operator<<(std::ostream& os, Figure& p)
	{
		p.Print(os);
		return os;
	}

	friend std::istream& operator >> (std::istream& is, Figure& p)
	{
		p.Read(is);
		return is;
	}
};

#endif
```

```
// main.cpp

#include <iostream>
#include "tQueue.h"
#include "tQueue.cpp"

using namespace std;

int main()
{
	TQueue<Rectangle> queue = TQueue<Rectangle>();
	for (int i = 0; i < 30; i++)
	{
		Rectangle *rct = new Rectangle();
		rct->apixes[0].x = i + 1;
		queue.Push(*rct);
		delete rct;
	}
	for (Figure *elem : queue)
	{
		cout << *elem << " " << endl;
	}
    Rectangle rect1, rect2;
    cin >> rect1;
    cin >> rect2;
    queue.Push(rect1);
    queue.Push(rect2);
    cout << queue.Length() << "  " << queue.Top();
    cout << queue;
```

```
    rect2.apixes[0].x = 42;
    queue.Push(rect2);
    cout << queue.Length() << "  " << queue.Top();
    cout << queue;
    queue.Pop();
    cout << queue.Length() << "  " << queue.Top();
    cout << queue;
    queue.Pop();
    cout << queue.Length() << "  " << queue.Top();
    cout << queue;
}
```

```
// point.h
#pragma once
#ifndef POINT_H
#define POINT_H

#include <iostream>

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

      double x;
      double y;

      double DistanceTo(Point& other);

      std::string ToString();

      Point& operator=(const Point& sq);
      bool operator==(const Point& sq);

      friend std::istream& operator>>(std::istream& is, Point& p);
      friend std::ostream& operator<<(std::ostream& os, Point& p);
};

#endif // POINT_H
```

```
//POINT.CPP
#include "point.h"

#include <cmath>
#include <string>


Point::Point() : x(0.0), y(0.0) {}

Point::Point(double xv, double yv) : x(xv), y(yv) {}

Point::Point(std::istream& is) {
      is >> x >> y;
}

double Point::DistanceTo(Point& other) {
      double dx = (other.x - x);
      double dy = (other.y - y);
      return std::sqrt(dx * dx + dy * dy);
```

```
}

std::string Point::ToString()
{
	return "(" + std::to_string(x) + ", " + std::to_string(y) + ")";
}

Point& Point::operator=(const Point& other)
{
	x = other.x;
	y = other.y;
	return *this;
}

bool Point::operator==(const Point& other)
{
	return x == other.x && y == other.y;
}

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;
}
```

```
// queueNode.cpp
#include "queueNode.h"
#define sNode std::shared_ptr<QueueNode<T>>

template<class T>
QueueNode<T>::QueueNode(const T& value)
{
	next = NULL;
	Data = value;
}

template<class T>
QueueNode<T>::QueueNode()
{
	next = NULL;
}
```

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

Rectangle::Rectangle(const Rectangle& other)
{
	figureName = "Rectangle";
	for (int i = 0; i < 4; i++)
	{
		apixes.push_back(other.apixes[i]);
	}
}
```

```
Rectangle::Rectangle()
{
	figureName = "Rectangle";
}

double Rectangle::Area()
{
	return apixes[0].DistanceTo(apixes[1])
		* apixes[1].DistanceTo(apixes[2]);
}

Rectangle::Rectangle(std::istream& inputStream)
{
	figureName = "Rectangle";
	for (int i = 0; i < 4; i++)
	{
		Point inputPoint(0, 0);
		inputStream >> inputPoint.x >> inputPoint.y;
		apixes.push_back(inputPoint);
	}

}

bool Rectangle::operator==(const Rectangle& other) const
{
	for (int i = 0; i < 4; i++)
	{
		if (apixes[i].x != other.apixes[i].x && apixes[i].y !=
other.apixes[i].y)
			return false;
	}
	return true;
}

std::istream& operator>>(std::istream& inputStream, Rectangle& rect)
{
	for (int i = 0; i < 4; i++)
	{
		Point inputPoint(0, 0);
		inputStream >> inputPoint.x >> inputPoint.y;
		rect.apixes.push_back(inputPoint);
	}
	return inputStream;
}

std::ostream& operator<<(std::ostream& outputStream, const Rectangle& rect)
{
	((Figure&)rect).Print(outputStream);
	return outputStream;
}
```

```
// rectangle.h
#pragma once
#ifndef RECTANGLE_H_INCLUDED
#define RECTANGLE_H_INCLUDED
#include "figure.h"
class Rectangle :
    public Figure
{
public:
```

```
    friend std::istream& operator>>(std::istream& is, Rectangle& rect);
    friend std::ostream& operator<<(std::ostream& os, const Rectangle& rect);

    Rectangle(const Rectangle&);
    Rectangle();
    double Area() override;
    Rectangle(std::istream&);
    bool operator==(const Rectangle& other) const;
};

#endif
```

```
#include "tQueue.h"
#include <string>
#define sNode std::shared_ptr<QueueNode<T>>


template<class T>
TQueue<T>::TQueue()
{
	size = 0;
	head = QueueNode<T>();
	last = &head;
}

template<class T>
TQueue<T>::TQueue(const TQueue<T>& other): TQueue()
{
	sNode current = other.head.next;
	while (current != NULL)
	{
		this->Push(current->Data);
		current = current->next;
	}
}

template<class T>
void TQueue<T>::Push(const T& value)
{
	last->next = sNode(new QueueNode<T>(value));
	last = last->next.get();
	size++;
}

template<class T>
void TQueue<T>::Pop()
{
	if (size == 0)
	{
		throw;
	}
	auto nextNode = head.next->next;
	head.next = NULL;
	head.next = nextNode;
	size--;
}

template<class T>
const T& TQueue<T>::Top()
{
	if (size == 0)
```

```
	{
		throw;
	}
	return head.next->Data;
}

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

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

template<class T>
void TQueue<T>::Print(std::ostream& os)
{
	sNode current = head.next;
	std::string output = "";
	while (current != NULL)
	{
		output = std::to_string(current->Data.Area()) + " " + output;
		current = current->next;
	}
	output = "=> " + output + "=>";
	os << output << std::endl;
}

template<class T>
void TQueue<T>::Clear()
{
	while (size != 0)
	{
		Pop();
	}
}

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

template<class T>
std::ostream& operator<<(std::ostream& os, TQueue<T>& queue)
{
	queue.Print(os);
	return os;
}
```

```
// tQueue.h
#pragma once
#ifndef TQUEUE_H_INCLUDED
#define TQUEUE_H_INCLUDED

#include "rectangle.h"
#include "queueNode.h"
```

```
#define sNode std::shared_ptr<QueueNode<T>>

template<class T>
class TQueue {
private:
	size_t size;
	QueueNode<T> head;
	QueueNode<T>* last;
public:
	TQueue();
	TQueue(const TQueue& other);
	void Push(const Rectangle& rectangle);
	void Pop();
	const Rectangle& Top();
	bool Empty();
	size_t Length();
	friend std::ostream& operator<<(std::ostream& os, const TQueue& queue);
	void Clear();
	virtual ~TQueue();
};

#endif
```

```
// queueNode.h
#pragma once
#ifndef QUEUE_NODE_H_INCLUDED
#define QUEUE_NODE_H_INCLUDED
#include "rectangle.h"
#include <memory>
#define sNode std::shared_ptr<QueueNode<T>>

template<class T>
class QueueNode
{
public:
	QueueNode(const T&);
	QueueNode();
	T Data;

	sNode next;
};

#endif
```

```
// titerator.h

#pragma once
//TITERATOR.H

#ifndef TITERATOR_H
#define TITERATOR_H

#include <memory>

template<class E> class TIterator
{
public:
	TIterator<E>(QueueNode<E> *node)
	{
		cur = node;
	}
```

```
	E* operator*()
	{
		return &(cur->Data);
	}

	E* operator->()
	{
		return &(cur->Data);
	}

	void operator++()
	{
		cur = cur->next.get();
	}

	TIterator<E> operator++(int)
	{
		TIterator iter(cur);
		++(*this);
		return iter;
	}

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

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

private:
	QueueNode<E> *cur;
};

#endif
```

```
//tvector.cpp
#include "stdafx.h"
#include "tvector.h"
#include <cstring>

TVector::TVector()
{
	vals = NULL;
	len = 0;
	rLen = 0;
}

void TVector::Erase(int pos)
{
	if (len == 1)
	{
		Clear();
		return;
	}
	for (int i = pos; i<len - 1; i++)
		vals[i] = vals[i + 1];
	len--;
	if (len == rLen >> 1)
	{
```

```
            resize(len);
            rLen = len;
      }
}

void TVector::InsertLast(const velem& elem)
{
      if (rLen)
      {
            if (len >= rLen)
            {
                  rLen <<= 1;
                  resize(rLen);
            }
      }
      else
      {
            rLen = 1;
            resize(rLen);
      }
      vals[len] = elem;
      len++;
}

velem& TVector::operator[](const size_t idx)
{
      return vals[idx];
}

bool TVector::Empty()
{
      return len == 0;
}

size_t TVector::Length()
{
      return len;
}

void TVector::Clear()
{
      if (!Empty())
      {
            delete[] vals;
            vals = NULL;
            len = 0;
            rLen = 0;
      }
}

void TVector::resize(int newsize)
{
      velem *newVals = new velem[newsize];
      for (int i = 0; i<len; i++)
            newVals[i] = vals[i];
      delete[] vals;
      vals = newVals;
}

TVector::~TVector()
{
      Clear();
```

```
}
```

```
// tvector.h
#pragma once

#ifndef TVECTOR_H
#define TVECTOR_H

#include <memory>

class velem
{
public:
      int *usedBy;
      void *value;
};

class TVector
{
      public:
            TVector();
            void Erase(int pos);
            void InsertLast(const velem& elem);
            velem& operator[](const size_t idx);
            void Clear();
            bool Empty();
            size_t Length();
            ~TVector();
      private:
            void resize(int newsize);
            velem *vals;
            int len;
            int rLen;
};

#endif
```

```
//tallocator.cpp
#include "stdafx.h"
#include "tallocator.h"

TAllocator::TAllocator(int elmSize, int bnchSize)
{
      elemSize = elmSize;
      bunchSize = bnchSize;
      allocated = new TVector();
      used = new TVector();
}

void* TAllocator::Allocate()
{
      if (allocated->Empty())
      {
            char *newBlock = (char*)malloc(sizeof(int)+elemSize*bunchSize);
            *(int*)(newBlock) = 0;
            for (int i = 0; i < bunchSize; i++)
            {
                  allocated->InsertLast(velem { (int*)newBlock, (void*)(newBlock +
sizeof(int) + i * elemSize) });
            }
      }
      int lastIdx = allocated->Length() - 1;
```

```
	void *block = ((*allocated)[lastIdx]).value;
	(*((*allocated)[lastIdx].usedBy))++;
	used->InsertLast((*allocated)[lastIdx]);
	allocated->Erase(lastIdx);
	return block;
}

void TAllocator::Deallocate(void* ptr)
{
	int fid = -1;
	for (int i = 0; i < used->Length(); i++)
	{
		if (((*used)[i]).value == ptr)
		{
			fid = i;
			break;
		}
	}
	(*((*used)[fid].usedBy))--;
	allocated->InsertLast((*used)[fid]);
	int *block = ((*used)[fid]).usedBy;
	if (*block == 0)
	{
		for (int i = allocated->Length()-1; i >= 0 ; i--)
		{
			if ((*allocated)[i].usedBy == block)
			{
				allocated->Erase(i);
			}
		}
		free(block);
	}
	used->Erase(fid);
}

TAllocator::~TAllocator()
{
	delete allocated;
	delete used;
}
```

```
// tallocator.h
#pragma once
#ifndef TALLOCATOR_H
#define TALLOCATOR_H

#include "tvector.h"

class TAllocator
{
public:
	TAllocator(int elmSize, int bunchSize);
	void* Allocate();
	void Deallocate(void* ptr);
	~TAllocator();

private:
	int elemSize;
	int bunchSize;
	TVector *allocated;
	TVector *used;
```

```
};

#endif
```