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: *Макаров Глеб Александрович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович*

Условие

Задание: Стэк (Пятиугольник). Используя структуру данных, разрабо- танную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных. Цель построения аллокатора – минимизация вы-зова 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.

Описание программы

Исходный код лежит в 15 файлах:

1. main.cpp: тестирование кода
2. figure.h: родительский класс-интерфейс для фигур
3. point.h: описание класса точки
4. point.cpp: реализация класса точки
5. pentagon.h: описание класса пятиугольника, наследующегося от figure
6. pentagon.cpp: реализация класса пятиугольника
7. template.cpp: файл для правильного подключения шаблонов класса.
8. tstack.h: структура стэка
9. tstack.cpp: реализация стэка
10. titerator.h: описание итераторов
11. titerator.cpp: реализация итераторов
12. tallocationblock.h: описание аллокатора

13. tallocationblock.cpp: реализация аллокатора
14. tlinkedlist.h: описание связанного списка
15. tlinkedlist.cpp: реализация связанного списка

Дневник отладки

Ошибок не было

Недочёты

Недочётов не заметил.

Вывод

В данной лабораторной работе были реализованы аллокаторы классов. Основной код уже был написан в предыдущих работах. Аллокаторы используются, когда необходимо придумать свои правила выделения памяти, а также снизить количество системных вызовов.

Исходный код

main.cpp

```
#include "tstack.h"


void Test()
{
    TAllocationBlock allocator(sizeof(int),10);

    int *b1=nullptr;
    int *b2=nullptr;
    int *b3=nullptr;
    int *b4=nullptr;
    int *b5=nullptr;

    b1 = (int*)allocator.Allocate();
    *b1 =1;
    std::cout << "b1 pointer value:" << *b1 << std::endl;
    b2 = (int*)allocator.Allocate();
    *b2 =2;
    std::cout << "b2 pointer value:" << *b2 << std::endl;
    b3 = (int*)allocator.Allocate();
    *b3 =3;
    std::cout << "b3 pointer value:" << *b3 << std::endl;

    allocator.Deallocate(b1);
    allocator.Deallocate(b3);

    b4 = (int*)allocator.Allocate();
    *b4 =4;
```

```
    std::cout << "b4 pointer value:" << *b4 << std::endl;
    b5 = (int*)allocator.Allocate();
    *b5 =5;
    std::cout << "b5 pointer value:" << *b5 << std::endl;
    std::cout << "b1 pointer value:" << *b1 << std::endl;
    std::cout << "b2 pointer value:" << *b2 << std::endl;
    std::cout << "b3 pointer value:" << *b3 << std::endl;

    allocator.Deallocate(b2);
    allocator.Deallocate(b4);
    allocator.Deallocate(b5);
}

int main() {
    Test();
    return 0;
}
```

figure.h

```
#ifndef MAI_OOP_FIGURE_H
#define MAI_OOP_FIGURE_H
#include "point.h"
#include <memory>

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

```
};

#endif //MAI_OOP_FIGURE_H
```

point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

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

  double dist(const Point &other);

  double get_x();
  double get_y();

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

 private:
  double x_;
  double y_;
};
```

```
#endif // POINT_H
```

point.cpp

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

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::get_x() {
  return x_;
}

double Point::get_y() {
  return y_;
}

double Point::dist(const 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;
}

bool Point::operator==(const Point &p) {
  if (this->x_ == p.x_ && this->y_ == p.y_) {
    return true;
  } else return false;
}

Point &Point::operator=(const Point &p) {
  if (this == &p) {
    return *this;
  }
  this->x_ = p.x_;
  this->y_ = p.y_;
  return *this;
}
```

pentagon.h

```
#ifndef MAI_OOP_PENTAGON_H
#define MAI_OOP_PENTAGON_H
#include "figure.h"

class Pentagon {
 private:
  Point a_, b_, c_, d_, e_;
 public:
  Pentagon();
     Pentagon(Point t_1, Point t_2, Point t_3, Point t_4,
          Point t_5);
  Pentagon(const Pentagon &pentagon);
  Pentagon(std::istream &is);
```

```
  size_t VertexesNumber();
  double Area();
  void Print(std::ostream &os);
  friend std::istream &operator>>(std::istream &is, Pentagon &object);
  friend std::ostream &operator<<(std::ostream &os, Pentagon &object);
  Pentagon &operator=(const Pentagon &object);
  bool operator==(const Pentagon &object);

};

#endif //MAI_OOP_PENTAGON_H
```

pentagon.cpp

```
#include "pentagon.h"
#include <math.h>

Pentagon::Pentagon() : a_(0, 0), b_(0, 0), c_(0, 0), d_(0, 0), e_(0, 0) {}

Pentagon::Pentagon(const Pentagon &pentagon) {
  this->a_ = pentagon.a_;
  this->b_ = pentagon.b_;
  this->c_ = pentagon.c_;
  this->d_ = pentagon.c_;
  this->e_ = pentagon.c_;
}

Pentagon::Pentagon(Point t_1, Point t_2, Point t_3, Point t_4,Point t_5)
    : a_(t_1), b_(t_2), c_(t_3), d_(t_4),
    e_(t_5) {}

Pentagon::Pentagon(std::istream &is) {
  std::cin >> a_ >> b_ >> c_ >> d_ >> e_;
}

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

double Pentagon::Area() {
  double p = fabs(a_.get_x()*b_.get_y()-b_.get_x()*a_.get_y()+b_.get_x()*c_.get_y()-
```

```
c_.get_x()*b_.get_x()+c_.get_x()*d_.get_y()-d_.get_x()*c_.get_y()+d_.get_x()*e_.get_y()-
e_.get_x()*d_.get_y()+e_.get_x()*a_.get_y()-a_.get_x()*e_.get_y())/2;
   return p;
}

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

std::istream &operator>>(std::istream &is, Pentagon &object) {
  is >> object.a_ >> object.b_ >> object.c_ >> object.d_ >> object.e_;
  return is;
}

std::ostream &operator<<(std::ostream &os,  Pentagon &object) {
  os << "a side = " << object.a_.dist(object.b_) << std::endl;
  os << "b side = " << object.b_.dist(object.c_) << std::endl;
  os << "c side = " << object.c_.dist(object.d_) << std::endl;
  os << "d side = " << object.d_.dist(object.e_) << std::endl;
  os << "e side = " << object.e_.dist(object.a_) << std::endl;
  return os;
}

Pentagon &Pentagon::operator=(const Pentagon &object) {
  this->a_ = object.a_;
  this->b_ = object.b_;
  this->c_ = object.c_;
  this->d_ = object.d_;
  this->e_ = object.e_;
  return *this;
}

bool Pentagon::operator==(const Pentagon &object) {
  if (this->a_ == object.a_ && this->b_ == object.b_ && this->c_ == object.c_ && this->d_
== object.d_ && this->e_ == object.e_) {
    return true;
  } else return false;
}
```

tstack.h

```
#ifndef MAI_OOP_TSTACK_H
#define MAI_OOP_TSTACK_H
#include "pentagon.h"
#include "titerator.h"
#include "tallocationblock.h"

template <class T>
class TStack {
 private:
  struct StackItem {
    std::shared_ptr<T> data;
    std::shared_ptr<StackItem> next;
  };
  size_t size;
  std::shared_ptr<StackItem> top_;

 public:
  TStack();
  TStack(const TStack<T> &stack);
  size_t Length();
  bool Empty();
  T Top();
  void Push(const std::shared_ptr<T> t);
  void Pop();
  void Clear();
  template<typename Y>
  friend std::istream &operator>>(std::istream &is, TStack<Y> &object);
  template<typename Y>
  friend std::ostream &operator<<(std::ostream &os, const TStack<Y> &object);
```

```
  TIterator<StackItem, T> top();

  virtual ~TStack();
};

#endif
```

tstack.cpp

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

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

Item* Item::to_right(Item* next) {
    Item* set = this->next;
    this->next = next;
    return set;
}

Item* Item::Next() {
    return this->next;
}

void* Item::GetItem() {
    return this->link;
}
```

```
Item::~Item() {}
```

template.h

```
#include "tstack.h"
#include "tstack.cpp"


template class TStack<Pentagon>;
template std::ostream& operator<< <Pentagon>(std::ostream&, TStack<Pentagon> const&);
```

tlinkedlist.h

```
#ifndef TLINKEDLIST_H
#define TLINKEDLIST_H

#include "item.h"

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

   void* GetItem();
```

```
    virtual ~TLinkedList();
private:
    Item* first;
};
#endif // TLINKEDLIST_H
```

tlinkedlist.cpp

```
#include "tlinkedlist.h"

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

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

void TLinkedList::Insert(size_t position, void *link) {
    Item *iter = this->first;
    auto *other = new Item(link);
    if (position == 1) {
        other->to_right(iter);
        this->first = other;
    } else {
        if (position <= this->Length()) {
            for (int i = 1; i < position - 1; ++i)
                iter = iter->Next();
```

```
            other->to_right(iter->Next());
            iter->to_right(other);
        }
    }
}

void TLinkedList::InsertLast(void *link) {
    auto *other = new Item(link);
    Item *iter = this->first;
    if (first != nullptr) {
        while (iter->Next() != nullptr) {
            iter = iter->to_right(iter->Next());
        }
        iter->to_right(other);
        other->to_right(nullptr);
    }
    else {
        first = other;
    }
}

size_t TLinkedList::Length() {
    size_t len = 0;
    Item* item = this->first;
    while (item != nullptr) {
        item = item->Next();
        len++;
    }
    return len;
}
```

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

void TLinkedList::Remove(size_t &position) {
    Item *iter = this->first;
    if (position <= this->Length()) {
        if (position == 1) {
            this->first = iter->Next();
        } else {
            int i = 1;
            for (i = 1; i < position - 1; ++i) {
                iter = iter->Next();
            }
            iter->to_right(iter->Next()->Next());
        }

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

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

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

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

item.h

```
#ifndef ITEM_H
#define ITEM_H

#include "pentagon.h"

class Item {
public:
    Item(void *ptr);

    Item* to_right(Item* next);
    Item* Next();
    void* GetItem();

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

#endif // ITEM_H
```

item.cpp

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

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

Item* Item::to_right(Item* next) {
    Item* set = this->next;
    this->next = next;
    return set;
}

Item* Item::Next() {
    return this->next;
}

void* Item::GetItem() {
    return this->link;
}

Item::~Item() {}
```

titerator.h

```
#ifndef TITERATOR_H
#define TITERATOR_H

#include <iostream>
#include <memory>

template <class node, class T>
```

```
class TIterator {
 public:
  TIterator(std::shared_ptr<node> n) { node_ptr = n; }

  std::shared_ptr<T> operator*() { return (node_ptr->data); }

  std::shared_ptr<T> operator->() { return (node_ptr->data); }

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

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

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

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

 private:
  std::shared_ptr<node> node_ptr;
};

#endif  // TITERATOR_H
```

tallocationblock.h

```
#ifndef TALLOCATIONBLOCK_H
#define TALLOCATIONBLOCK_H

#include <iostream>
```

```
#include "TLinkedList.h"

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

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

   virtual ~TAllocationBlock();

private:
   char *used;
   TLinkedList unused;
};

#endif //TALLOCATIONBLOCK_H
```

tallocationblock.cpp

```
#include "tallocationblock.h"

TAllocationBlock::TAllocationBlock(size_t size, size_t count) {
   used = (char *)malloc(size * count);
   for (size_t i = 0; i < count; ++i) {
      void *ptr = (void *)malloc(sizeof(void *));
      ptr = used + i * size;
      unused.InsertLast(ptr);
   }
```

```
}

void *TAllocationBlock::Allocate() {
    if (!unused.Empty()) {
        void *res = unused.GetItem();
        size_t first = 1;
        unused.Remove(first);
        std::cout << "Pentagon created" << std::endl;
        return res;
    } else {
        throw std::bad_alloc();
    }
}

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

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

size_t TAllocationBlock::Size() {
    return unused.Length();
}

TAllocationBlock::~TAllocationBlock() {
    while (!unused.Empty()) {
        size_t first = 1;
        unused.Remove(first);
```

```
    }
    free(used);
    std::cout << "Pentagon deleted" << std::endl;
}
```