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 *Ханнанов Руслан Маратович, группа М8О-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

Условие

Задание: Вариант 22: 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

Исходный код лежит в 14 файлах:

1. src/main.cpp: основная программа, взаимодействие с пользователем посредством комманд из меню
2. include/figure.h: описание абстрактного класса фигур
3. include/point.h: описание класса точки
4. include/node.h: описание класса ноды дерева
5. include/pentagon.h: описание класса пятиугольника, наследующегося от figures
6. include/tnary_tree.h: описание класса N-дерева
7. include/iterator.h: описание класса итератора
8. include/list_item.h: описание класса элемента списка
9. include/tallocator.h: описание класса аллокатора
10. include/tlinked_list.h: описание класса связного списка
11. include/point.cpp: реализация класса точки
12. include/pentagon.cpp: реализация класса пятиугольника, наследующегося от figures
13. include/node.cpp: реализация класса ноды дерева
14. include/tnary_tree.cpp: реализация класса N-дерева

Дневник отладки

Были трудности с реализацией аллокатора.

Недочёты

Недочетов не обнаружено.

Выводы

В процессе выполнения данной лабораторной работы я познакомился с понятием аллокатора. Аллокаторы - очень важная часть любой структуры данных, поэтому, как мне кажется, знакомство с ней необходимо каждому программисту.

Исходный код:

iterator.h

```
//
// Created by Руслан on 27.12.2021.
//

#ifndef LAB6__ITERATOR_H_
#define LAB6__ITERATOR_H_
#include <iostream>
#include <memory>

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

  T operator*() {
    return *(node_ptr->key);
  }

  void to_son(){
    if (node_ptr->son == nullptr){
      std::cout << "Node does not exist" << std::endl;
    } else {
      node_ptr = node_ptr->son;
    }
  }
  void to_brother(){
    if (node_ptr->brother == nullptr){
      std::cout << "Node does not exist" << std::endl;
    } else {
      node_ptr = node_ptr->brother;
    }
  }

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

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

 private:
  std::shared_ptr<item> node_ptr;
};
#endif //LAB6__ITERATOR_H_
```

figure.h

```
#ifndef LAB6__FIGURE_H_
#define LAB6__FIGURE_H_
#include <memory>
#include "point.h"

class Figure {
 private:
  virtual void Print(std::ostream& os) = 0;
  virtual double Area() = 0;
  virtual size_t VertexesNumber() = 0;
};
#endif //LAB6__FIGURE_H_
```

point.h

```
#ifndef LAB6__POINT_H_
#define LAB6__POINT_H_
#include <istream>

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

  double dist(Point& other);
  friend double getx(Point& p);
  friend double gety(Point& p);

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

```
 private:
  double x_;
  double y_;

};
#endif //LAB6__POINT_H_
```

point.cpp

```
//
// Created by Руслан on 27.12.2021.
//

#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::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, const Point& p) {
  os << "(" << p.x_ << ", " << p.y_ << ") ";
  return os;
}

double getx(Point& p) {
  return p.x_;
```

```
}

double gety(Point& p) {
  return p.y_;
}
```

pentagon.h

```
//
// Created by Руслан on 27.12.2021.
//

#ifndef LAB6__PENTAGON_H_
#define LAB6__PENTAGON_H_
#include "figure.h"
#include "point.h"
#include <iostream>
#include "tallocator.h"

class Pentagon : public Figure {
 public:
  Pentagon();
  Pentagon(Point v1,Point v2,Point v3,Point v4,Point v5);
  explicit Pentagon(std::istream &is);
  Pentagon(Pentagon &other);

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

  friend std::ostream &operator<<(std::ostream &os, const Pentagon &figure);

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

  ~Pentagon();
 private:
  static TAllocator allocator;
  Point v1,v2,v3,v4,v5;
};
#endif //LAB6__PENTAGON_H_
```

pentagon.cpp

```cpp
//
// Created by Руслан on 27.12.2021.
//

#include "pentagon.h"
#include <cmath>
#include "tallocator.h"

Pentagon::Pentagon(): v1(0,0),v2(0,0),v3(0,0),v4(0,0),v5(0,0){
  //std::cout << "Default pentagon created" << std::endl;
}

Pentagon::Pentagon(Point v_1,Point v_2, Point v_3, Point v_4, Point v_5):
    v1(v_1), v2(v_2), v3(v_3), v4(v_4), v5(v_5)
{
  //std::cout << "Pentagon created" << std::endl;
}

Pentagon::Pentagon(Pentagon& other):
    Pentagon(other.v1,other.v2,other.v3,other.v4,other.v5)
{
  //std::cout << "Made copy of pentagon";
}

Pentagon::Pentagon(std::istream &is) {
  is >> v1 >> v2 >> v3 >> v4 >> v5;
  //std::cout << "Pentagon created via istream" << std::endl;
}

void Pentagon::Print(std::ostream& os) {
  os << "Pentagon:" << v1 << v2 << v3 << v4 << v5 << "\n";
}

Pentagon::~Pentagon() {
  //std::cout << "Object Pentagon ";
  //Print(std::cout);
  //std::cout << "deleted" << std::endl;
}

double Pentagon::Area() {
  Point ar[5];
  ar[0] = v1;
```

```
  ar[1] = v2;
  ar[2] = v3;
  ar[3] = v4;
  ar[4] = v5;
  double res = 0;
  for (unsigned i = 0; i < 5; i++) {
    Point p = i ? ar[i-1] : ar[4];
    Point q = ar[i];
    res += (getx(p) - getx(q)) * (gety(p) + gety(q));
  }
  return fabs(res) / 2;
}

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

TAllocator Pentagon::allocator(sizeof(Pentagon), 20);

void* Pentagon::operator new(size_t size) {
  return allocator.allocate(size);
}

void Pentagon::operator delete(void *ptr) {
  allocator.deallocate(ptr);
}

std::ostream &operator<<(std::ostream &os, const Pentagon &figure)
{
  os << "Pentagon: " << figure.v1 << " " << figure.v2 << " " << figure.v3 << " " << figu
  return os;
}
```

node.h

```
//
// Created by Руслан on 27.12.2021.
//

#ifndef LAB6__NODE_H_
#define LAB6__NODE_H_
#include <iostream>
#include <memory>
```

```
#include "pentagon.h"
template <class T>

class Node {
 public:

  Node(const std::shared_ptr<T> &k);
  Node(const Node &other);

  std::shared_ptr<Node<T>> last_son();

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

  std::shared_ptr<Node<T>> son;
  std::shared_ptr<Node<T>> brother;
  std::shared_ptr<T> key;
};
#endif //LAB6__NODE_H_
```

node.cpp

```
#include "node.h"

template <class T>
Node<T>::Node(const std::shared_ptr<T> &k) {
  key = k;
  son = nullptr;
  brother = nullptr;
}

template <class T>
Node<T>::Node(const Node &other) {
  this->son = other.son;
  this->brother = other.brother;
  this->key = other.key;
}

template <class T>
std::shared_ptr<Node<T>> Node<T>::last_son() {
  std::shared_ptr<Node<T>> ls = son;
  int number_of_sons = 0;
  while (ls->brother != nullptr) {
```

```
    ls = ls->brother;
    ++number_of_sons;
  }
  return ls;
}

template<class T>
std::ostream &operator<<(std::ostream & os, const Node<T> &obj) {
  os << "Item: " << *obj.key << std::endl;
  return os;
}

#include "pentagon.h"
template class Node<Pentagon>;
template std::ostream& operator<<(std::ostream& os, const Node<Pentagon>& obj);
```

tnary_tree.h

```
//
// Created by Руслан on 27.12.2021.
//

#ifndef LAB6__TNARY_TREE_H_
#define LAB6__TNARY_TREE_H_
#include "node.h"
#include "iterator.h"
template <class T>
class TNaryTree {
 public:
  TNaryTree();
  TNaryTree(int n);
  TNaryTree(const TNaryTree<T> &other);

  void Update(std::shared_ptr<T> k, const std::string &&tree_path = "");
  void RemoveSubTree(const std::string &&tree_path);
  bool Empty();

  template<class A>
  friend std::ostream &operator<<(std::ostream &os, const TNaryTree<A> &tree);

  std::shared_ptr<Node<T>> getroot();

  const std::shared_ptr<T> &GetItem(const std::string &&tree_path = "");
```

```
  virtual ~TNaryTree();

 private:
  int N;
  std::shared_ptr<Node<T>> root;

  void sub_copy(std::shared_ptr<Node<T>> &cur, std::shared_ptr<Node<T>> cp);
  void set_root(std::shared_ptr<T> &k);
  void sub_remove(std::shared_ptr<Node<T>> cur);
  void sub_print_operator(std::shared_ptr<Node<T>> cur, std::ostream &os) const;
};
#endif //LAB6__TNARY_TREE_H_
```

tnary_tree.cpp

```
//
// Created by Руслан on 27.12.2021.
//

#include "tnary_tree.h"

template <class T>
TNaryTree<T>::TNaryTree(int n) {
  N = n;
  root = nullptr;
}

template <class T>
TNaryTree<T>::TNaryTree() {
  N = 3;
  root = nullptr;
}

template <class T>
void TNaryTree<T>::Update(std::shared_ptr<T> k, const std::string &&tree_path) {
  if (tree_path == "") {
    if (root == nullptr) {
      set_root(k);
    } else root->key = k;
  } else {
    std::shared_ptr<Node<T>> cur = root;
    int degree = 1;
    for (int i = 0; i < tree_path.size() - 1; ++i) {
```

```
      if (tree_path[i] == 'c') {

        degree = 1;
        cur = cur->son;
        if (cur == nullptr) {
          throw std::invalid_argument("The node doesn't exist");
        }

      }
      if (tree_path[i] == 'b') {

        ++degree;
        cur = cur->brother;
        if (cur == nullptr) {
          throw std::invalid_argument("The node doesn't exist");
        }

      }

    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      if (cur->son == nullptr) {
        std::shared_ptr<Node<T>> item(new Node(k));
        cur->son = item;
      } else cur->son->key = k;
    }
    if (tree_path[tree_path.size() - 1] == 'b') {
      ++degree;
      if (degree > N) {
        throw std::out_of_range("Node cannot be added due to overflow");
      }
      if (cur->brother == nullptr) {
        std::shared_ptr<Node<T>> item(new Node(k));
        cur->brother = item;
      } else cur->brother->key = k;
    }
  }
}

template <class T>
void TNaryTree<T>::set_root(std::shared_ptr<T> &k) {
```

```
  std::shared_ptr<Node<T>> cur(new Node(k));
  root = cur;
}

template <class T>
bool TNaryTree<T>::Empty() {
  if (root == nullptr)
    return true;
  else return false;
}

template <class T>
void TNaryTree<T>::RemoveSubTree(const std::string &&tree_path) {
  std::shared_ptr<Node<T>> cur;
  if (tree_path == "") {
    sub_remove(root);
  } else {
    cur = root;
    for (int i = 0; i < tree_path.size() - 1; ++i) {

      if (tree_path[i] == 'c') {

        cur = cur->son;

      }
      if (tree_path[i] == 'b') {

        cur = cur->brother;

      }

    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      std::shared_ptr<Node<T>> cur_d = cur->son;
      cur->son = nullptr;
      sub_remove(cur_d);
    }
    if (tree_path[tree_path.size() - 1] == 'b') {
      std::shared_ptr<Node<T>> cur_d = cur->brother;
      cur->brother = nullptr;
      sub_remove(cur_d);
    }
```

```
  }
}

template <class T>
void TNaryTree<T>::sub_remove(std::shared_ptr<Node<T>> cur) {
  if (cur->son != nullptr) {
    std::shared_ptr<Node<T>> it = cur->son;
    cur->son = nullptr;
    sub_remove(it);
  }
  if (cur->brother != nullptr) {
    std::shared_ptr<Node<T>> it = cur->brother;
    cur->brother = nullptr;
    sub_remove(it);
  }
  cur.reset();
  return;
}

template <class T>
TNaryTree<T>::~TNaryTree() {
  if (root == nullptr) {
    return;
  }
  sub_remove(root);
}

template <class T>
TNaryTree<T>::TNaryTree(const TNaryTree &other) {
  N = other.N;
  std::shared_ptr<Node<T>> cp = other.root;
  std::shared_ptr<Node<T>> item(new Node(cp->key));
  root = item;
  std::shared_ptr<Node<T>> cur = root;
  if (cp->son != nullptr) {
    sub_copy(cur->son, cp->son);
  }
  if (cp->brother != nullptr) {
    sub_copy(cur->brother, cp->brother);
  }
}
```

```cpp
template <class T>
void TNaryTree<T>::sub_copy(std::shared_ptr<Node<T>> &cur, std::shared_ptr<Node<T>> cp)
  std::shared_ptr<Node<T>> item(new Node(cp->key));
  cur = item;
  if (cp->son != nullptr) {
    sub_copy(cur->son, cp->son);
  }
  if (cp->brother != nullptr) {
    sub_copy(cur->brother, cp->brother);
  }
  return;
}

template <class T>
std::ostream &operator<<(std::ostream &os, const TNaryTree<T> &tree) {
  std::shared_ptr<Node<T>> cur = tree.root;
  tree.sub_print_operator(cur, os);
  return os;
}

template <class T>
void TNaryTree<T>::sub_print_operator(std::shared_ptr<Node<T>> cur, std::ostream &os) co
  T k = *(cur->key);
  os << k.Area();
  if (cur->son != nullptr) {
    os << ": [";
    sub_print_operator(cur->son, os);
    os << "]";
  }
  if (cur->brother != nullptr) {
    os << ", ";
    sub_print_operator(cur->brother, os);
    //os << "]";
  }
  return;
}

template <class T>
const std::shared_ptr<T> &TNaryTree<T>::GetItem(const std::string &&tree_path) {
  std::shared_ptr<Node<T>> cur;
  if (tree_path == "") {
    return root->key;
```

```
  } else {
    cur = root;
    for (int i = 0; i < tree_path.size() - 1; ++i) {

      if (tree_path[i] == 'c') {

        cur = cur->son;

      }
      if (tree_path[i] == 'b') {

        cur = cur->brother;

      }

    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      return cur->son->key;
    }
    if (tree_path[tree_path.size() - 1] == 'b') {
      return cur->brother->key;
    }
  }
}

template <class T>
std::shared_ptr<Node<T>> TNaryTree<T>::getroot(){
  return root;
};

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

list_item.h

```
#ifndef LAB6__LIST_ITEM_H_
#define LAB6__LIST_ITEM_H_

#include <memory>

template <typename T>
class ListItem {
 public:
```

```
  ListItem(): data(0) {}
  ListItem(T t): data(t){}
  std::shared_ptr<ListItem<T>> getNext() {
    return next;
  }

  void setNext(std::shared_ptr<ListItem<T>> _next) {
    next = _next;
  }

  T getData() {
    return data;
  }
 private:
  std::shared_ptr<ListItem<T>> next = nullptr;
  T data;
};


#endif //LAB6__LIST_ITEM_H_
```

tlinked_list.h

```
#ifndef LAB6__TLINKED_LIST_H_
#define LAB6__TLINKED_LIST_H_

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

template<typename T>
class TLinkedList {
 public:
  TLinkedList() : length(0), head(nullptr) {}

  void pushBack(T t) {
    if (length == 0) {
      head = std::make_shared<ListItem<T>>(new (ListItem<T>));
    } else {
      ListItem<T> temp;
      while (temp.getNext() != nullptr) {
        temp = *temp.getNext();
      }
```

```
    }
    length++;
  }

  int size() {
    return length;
  }

  void erase(int i) {
    while (i--) {
      head = head->getNext();
    }
    if (length > 0) {
      length--;
    } else {
      std::cout << "list too small" << std::endl;
    }
  }

  std::shared_ptr<ListItem<T>> getFirst() {
    return head;
  }

 private:
  std::shared_ptr<ListItem<T>> head;
  int length;
};


#endif //LAB6__TLINKED_LIST_H_
```

tallocator.h

```
#ifndef LAB6__TALLOCATOR_H_
#define LAB6__TALLOCATOR_H_

#include "tlinked_list.h"

class TAllocator {
 public:
  TAllocator(size_t size, size_t count) {
    this->size = size;
    for (int i = 0; i < count; i++) {
```

```
    freeBlocks.pushBack(malloc(size));
  }
}

void* allocate(size_t size) {
  if (size != this->size) {
    std::cout << "Cant find such memory" << std::endl;
  }
  if (freeBlocks.size() != 0) {
    for (int i = 0; i < 5; i++) {
      freeBlocks.pushBack(malloc(size));
    }
  }

  void* temp = freeBlocks.getFirst()->getData();
  usedBlocks.pushBack(freeBlocks.getFirst()->getData());
  freeBlocks.erase(0);
  return temp;
}

void deallocate(void* ptr) {
  freeBlocks.pushBack(ptr);
}

~TAllocator() {
  while (usedBlocks.size() != 0) {
    try {
      free(usedBlocks.getFirst()->getData());
      usedBlocks.erase(0);
    } catch(...) {
      usedBlocks.erase(0);
    }
  }
  while (freeBlocks.size() != 0) {
    try{
      free(freeBlocks.getFirst()->getData());
      freeBlocks.erase(0);
    } catch(...) {
      freeBlocks.erase(0);
    }
  }
}
```

```
 private:
  size_t size;
  TLinkedList<void*> usedBlocks;
  TLinkedList<void*> freeBlocks;
};

#endif //LAB6__TALLOCATOR_H_
```

main.cpp

```
#include <iostream>
#include "pentagon.h"
#include "tnary_tree.h"
int main() {
  TNaryTree<Pentagon> t(3);

  std::shared_ptr<Pentagon> p1(new Pentagon({0, 0}, {0, 2}, {1, 3}, {2, 3}, {3, 0}));
  std::shared_ptr<Pentagon> p2(new Pentagon({0, 0}, {0, 3}, {1, 3}, {2, 3}, {3, 0}));
  std::shared_ptr<Pentagon> p3(new Pentagon({0, 0}, {0, 4}, {1, 3}, {2, 3}, {3, 0}));
  std::shared_ptr<Pentagon> p4(new Pentagon({0, 0}, {0, 6}, {1, 3}, {2, 3}, {3, 0}));
  std::shared_ptr<Pentagon> p5(new Pentagon({0, 0}, {0, 8}, {1, 3}, {2, 3}, {3, 0}));
  std::shared_ptr<Pentagon> p6(new Pentagon({0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}));

  t.Update(p1,"");
  t.Update(p2,"c");
  t.Update(p3,"cb");
  t.Update(p4,"cbb");
  t.Update(p5,"cc");
  std::cout << t << std::endl;

  std::shared_ptr<Node<Pentagon>> r = t.getroot();

  TIterator<Node<Pentagon>, Pentagon> iter1(r);

  std::cout << "Iterator work:" << std::endl;
  std::cout << *iter1 << std::endl;
  iter1.to_son();
  std::cout << *iter1 << std::endl;
  iter1.to_brother();
  std::cout << *iter1 << std::endl;

  TIterator<Node<Pentagon>, Pentagon> iter2(r);
```

```
  iter2.to_son();

  return 0;
}
```