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

ЛАБОРАТОРНАЯ РАБОТА №8

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

Студент *Маринин Иван Сергеевич, группа М80-208Б-20*
Преподаватель *Дорохов Евгений Павлович*

Условие

Задание: Вариант 10: Трапеция, Квадрат, Прямоугольник. 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 файл лежит в 15 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством комманд из меню
2. figure.h: описание абстрактного класса фигур
3. square.h: описание класса квадрата, наследующегося от figures
4. pryam.p: описание класса прямоугольника, наследующегося от figures
5. trap.p: описание класса трапеции, наследующегося от figures
6. square.cpp: реализация класса квадрата, наследующегося от figures
7. pryam.cpp: реализация класса прямоугольника, наследующегося от figures
8. trap.cpp: реализация класса трапеции, наследующегося от figures
9. TNode.h: описание класса элемента связанного списка
10. TList.hpp: реализация класса элемента связанного списка

11. TList.cpp: реализация класса связанного списка
12. TList.h: описание класса связанного списка
13. TAllocator.h: описание класса аллокатора связного списка
14. TAllocator.cpp: реализация класса аллокатора связного списка
15. TIterator.hpp : реализация итератора

Дневник отладки

Возникли небольшие проблемы при реализации аллокатора. Но позже они были устранены.

```
ivanmarinin@Air-Ivan lab_6 % ./a.out
1) Add item to list
2) Print list
3) Delete item from list
4) Exit
1
Enter 1 if square, 2 if trap, 3 if pryam
1
1 3
Item was added
1) Add item to list
2) Print list
3) Delete item from list
4) Exit
1
Enter 1 if square, 2 if trap, 3 if pryam
1
2 5
Item was not added
1) Add item to list
2) Print list
3) Delete item from list
4) Exit
2
Type of figure is square
a = 1
1) Add item to list
2) Print list
3) Delete item from list
4) Exit
4
Bye!
ivanmarinin@Air-Ivan lab_6 %
```

Недочёты

Недочётов не было обнаружено.

Выводы

В ходе лабораторной работы №8 я ознакомился с понятием аллокаторов в языке C++, а также отточил навыки их использования. Я понял, что аллокаторы используются почти во всех структурах данных, поэтому это тема очень важна для любого программиста.

Исходный код

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H
#include <iostream>
#include <cmath>

class Figure {
 public:
  virtual double getSquare() = 0;
  virtual void Print() = 0;
  virtual ~Figure() {};
};

#endif
```

main.cpp

```
#include "TList.h"
#include <iostream>
#include "square.h"
#include "trap.h"
#include "pryam.h"

void menu(void) {
  std::cout << "1) Add item to list" << std::endl;
  std::cout << "2) Print list" << std::endl;
  std::cout << "3) Delete item from list" << std::endl;
  std::cout << "4) Exit" << std::endl;
}

int main(void) {
  TList<Figure> list;
  int opt, index;
```

```
Square tmp1;
Trap tmp2;
Pryam tmp3;

do {
  menu();
  std::cin >> opt;
  switch(opt) {
  case 1:{
    std::cout << "Enter 1 if square, 2 if trap, 3 if pryam" << std::endl;
    std::cin >> index;
    if (index == 1) {
      std::cin >> tmp1 >> index;
      if (list.Push(std::make_shared<Square>(tmp1), index))
        std::cout << "Item was added" << std::endl;
      else
        std::cout << "Item was not added" << std::endl;
      break;
    } else if (index == 2) {
      std::cout << "Enter value and index" << std::endl;
      tmp2.setParams(std::cin);
      std::cin >> index;
      if (list.Push(std::make_shared<Trap>(tmp2), index))
        std::cout << "Item was added" << std::endl;
      else
        std::cout << "Item was not added" << std::endl;
      break;
    } else if (index == 3) {
      std::cout << "Enter value and index" << std::endl;
      tmp3.setParams(std::cin);
      std::cin >> index;
      if (list.Push(std::make_shared<Pryam>(tmp3), index))
        std::cout << "Item was added" << std::endl;
      else
        std::cout << "Item was not added" << std::endl;
      break;
    } else {
      std::cout << "derp" << std::endl;
      break;
    }
  }
  case 2:
    for (const auto& i : list) {
      i->Print();
    }
    break;
  case 3:{
    std::cout << "Enter index" << std::endl;
    std::cin >> index;
    if (list.Pop(index))
      std::cout << "Item was removed" << std::endl;
    else
      std::cout << "Item was not removed" << std::endl;
    break;
  }
  }
} while(opt != 4);

  std::cout << "Bye!" << std::endl;
```

```
  return 0;
}
```

square.cpp

```
#include "square.h"

Square::Square() : Square(-1.0){}
Square::Square(const Square &obj) {
  side_a = obj.side_a;
}

Square::Square(double i) {
  this->side_a = i;
}

Square::Square(std::istream &is) {
  is >> this->side_a;
}

double Square::getSquare() {
  return this->side_a * this->side_a;
}

void Square::setParams(std::istream &is) {
  is >> this->side_a;
}

void Square::Print() {
  std::cout << "Type of figure is square" << std::endl
            << "a = " << this->side_a << std::endl;
}

Square Square::operator++() {
  this->side_a++;
}

Square Square::operator+(const Square& obj) const{
  Square res;
  res.side_a = side_a + obj.side_a;
  return res;
}

std::ostream& operator<<(std::ostream& os, const Square& obj) {
  if (obj == Square())
    return os;

  os << "Length of square is "
     << obj.side_a << std::endl;
  return os;
}

std::istream& operator>>(std::istream& is, Square& obj) {
  is >> obj.side_a;
  return is;
}

bool Square::operator==(const Square& obj) const{
  return side_a == obj.side_a;
}
```

```
Square Square::operator=(const Square& obj) {
  if (this == &obj) return *this;

  side_a = obj.side_a;
  return *this;
}
```

square.h

```
#ifndef SQUARE_H
#define SQUARE_H
#include <iostream>
#include "figure.h"

class Square: public Figure {
 public:
  Square();
  Square(const Square &obj);
  Square(double i);
  Square(std::istream &is);

  Square operator++();
  Square operator+(const Square& obj) const;
  friend std::ostream& operator<<(std::ostream& os, const Square& obj);
  friend std::istream& operator>>(std::istream& is, Square& obj);
  bool operator==(const Square& obj) const;
  Square operator=(const Square& obj);

  double getSquare() override;
  void Print() override;
  void setParams(std::istream &is);
  ~Square() {};

 private:
  double side_a;
};

#endif
```

TAllocator.cpp

```
#include "TAllocator.h"

TAllocator::TAllocator(const size_t& size, const size_t& amountToAdd) {
  sizeOfBlock = size;
  amount = amountToAdd;
  root->N = amount;
  root->sons = (TNTree*)malloc(sizeof(TNTree) * root->N);
  usedBlocks = (char*)malloc(sizeof(char) * sizeOfBlock * amount);

  for (size_t i = 0; i < amount; i++) {
    root->sons[i].data = malloc(sizeOfBlock);
    root->sons[i].data = usedBlocks + i * sizeOfBlock;
  }

  free = amountToAdd;
}
```

```
void* TAllocator::allocate() {
  void* result = nullptr;

  if (free > 0) {
    result = root->sons[free - 1].data;
    free--;
  } else {
    throw std::runtime_error("TAllocator: out of memory");
  }
}

void TAllocator::deallocate(void* ptr) {
  root->sons[free].data = ptr;
  free++;
}

TAllocator::~TAllocator() {
  if (free < amount) {
    throw std::runtime_error("TAllocator: memory leak");
  }

  std::free(usedBlocks);
  for (size_t i = 0; i < root->N - 1; i++) {
    std::free(root->sons[i].data);
  }
  std::free(root->sons);
}
```

TAllocator.h

```
#ifndef TALLOCATOR_H
#define TALLOCATOR_H

#include <cstdlib>
#include <stdexcept>

struct TNTree {
  TNTree* sons;
  void* data;
  size_t N;
};

class TAllocator {
 private:
  size_t free;
  size_t amount;
  size_t sizeOfBlock;

  TNTree* root;
  char* usedBlocks;
 public:
  TAllocator(const size_t&, const size_t&);
  void* allocate();
  void deallocate(void*);
  ~TAllocator();
};
```

```
#endif
```

TList.cpp

```
#include "TList.h"

TList::TNode::TNode() {
  item = std::make_shared<Square>();
  next = nullptr;
}

TList::TNode::TNode(const std::shared_ptr<Figure>& obj) {
  item = obj;
  next = nullptr;
}

TList::TList() {
  head = std::make_shared<TNode>();
  length = 0;
}

bool TList::IsEmpty() const {
  return this->length == 0;
}

int TList::GetLength() const {
  return this->length;
}

bool TList::PushFront(const std::shared_ptr<Figure>& obj) {
  auto Nitem = std::make_shared<TNode>(obj);
  std::swap(Nitem->next, head->next);
  std::swap(head->next, Nitem);
  length++;

  return true;
}

bool TList::Push(const std::shared_ptr<Figure>& obj, int pos = 1) {
  if (pos == 1 || length == 0)
    return PushFront(obj);
  if (pos < 0 || pos > length)
    return false;

  auto iter = head->next;
  int i = 0;

  while (i < pos - 2) {
    iter = iter->next;
    i++;
  }

  auto Nitem = std::make_shared<TNode>(obj);
  std::swap(Nitem->next, iter->next);
  std::swap(iter->next, Nitem);
  length++;

  return true;
}
```

```
bool TList::PopFront() {
  if (IsEmpty())
    return false;

  head->next = std::move(head->next->next);

  length--;

  return true;
}

bool TList::Pop(int pos = 1) {
  if (pos < 1 || pos > length || IsEmpty())
    return false;
  if (pos == 1)
    return PopFront();

  auto iter = head->next;
  int i = 0;

  while (i < pos - 2) {
    iter = iter->next;
    i++;
  }

  iter->next = std::move(iter->next->next);
  length--;

  return true;
}

auto TList::TNode::GetNext() const {
  return this->next;
}

auto TList::TNode::GetItem() const {
  return this->item;
}

std::ostream& operator<< (std::ostream& os, const TList& list) {
  if (list.IsEmpty()) {
    os << "The list is empty!" << std::endl;
    return os;
}

  auto tmp = list.head->GetNext();
  while(tmp != nullptr) {
    tmp->GetItem()->Print();
    tmp = tmp->GetNext();
  }

  return os;
}
```

TList.h

```
#ifndef TLIST_H
#define TLIST_H
```

```
#include <memory>
#include <iostream>
#include "TAllocator.h"

template <typename T> class TList {
 private:
  class TNode {
  public:
    TNode();
    TNode(const std::shared_ptr<T>&);
    auto GetNext() const;
    auto GetItem() const;
    std::shared_ptr<T> item;
    std::shared_ptr<TNode> next;

    void* operator new(size_t);
    void operator delete(void*);
    static TAllocator nodeAllocator;
  };

  template <typename N, typename M>
    class TIterator {
  private:
    N nodePtr;
  public:
    TIterator(const N&);
    std::shared_ptr<M> operator* ();
    std::shared_ptr<M> operator-> ();
    void operator ++ ();
    bool operator == (const TIterator&);
    bool operator != (const TIterator&);
  };

  int length;

  std::shared_ptr<TNode> head;

 public:
  TList();
  bool PushFront(const std::shared_ptr<T>&);
  bool Push(const std::shared_ptr<T>&, const int);
  bool PopFront();
  bool Pop(const int);
  bool IsEmpty() const;
  int GetLength() const;

  TIterator<std::shared_ptr<TNode>, T> begin() {return
TIterator<std::shared_ptr<TNode>, T>(head->next);};
  TIterator<std::shared_ptr<TNode>, T> end() {return
TIterator<std::shared_ptr<TNode>, T>(nullptr);};

  template <typename A> friend std::ostream& operator<< (std::ostream&,
TList<A>&);
};

#include "TList.hpp"
#include "TIterator.hpp"
#endif
```

TNode.h

```
#ifndef TNODE_H
#define TNODE_H
#include "TList.h"

class TNode {
 private:
  friend TList;
  Square keyS;
  Trap keyT;
  Pryam keyP;
  std::shared_ptr<TNode> next;
 public:
  TNode();
  TNode(const Square&, const Trap&, const Pryam&);
  void GetNext() const;
  friend std::ostream& operator<< (std::ostream&, const TNode&);
};

#endif
```