Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы

по курсу

Объектно-ориентированное программирование

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №1

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

- Программирование классов на языке C++
- Управление памятью в языке C++
- Изучение базовых понятий ООП.
- Знакомство с классами в C++.
- Знакомство с перегрузкой операторов.
- Знакомство с дружественными функциями.
- Знакомство с операциями ввода-вывода из стандартных библиотек.

Задание

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов задания.

Классы должны удовлетворять следующим правилам:

- Должны иметь общий родительский класс Figure.
- Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в стандартный поток вывода cout.
- Должный иметь общий виртуальный метод расчета площади фигуры – Square.
- Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного потока cin.
- Должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Программа должна позволять вводить фигуру каждого типа с клавиатуры, выводить параметры фигур на экран и их площадь.

Теория

Классы и объекты в C++ являются основными концепциями объектно-ориентированного программирования.

Классы в C++ — это абстракция описывающая методы, свойства, присущие сразу группе объектов.

Объекты — конкретное представление абстракции, имеющее свои свойства. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса. В ООП существует три основных принципа построения классов:

Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

Наследование — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.

Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Перегрузка — это возможность поддерживать несколько функций с одним названием, но разными сигнатурами вызова.

Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях `private` или `protected`

Листинг:

- `Figure.h`

```
#ifndef FIGURE_H
#define     FIGURE_H

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

#endif      /* FIGURE_H */
```

- Foursquare.h

```
#ifndef FOURSQUARE_H
#define     FOURSQUARE_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Foursquare : public Figure{
public:
    Foursquare();
    Foursquare(std::istream &is);
    Foursquare(size_t i);
    Foursquare(const Foursquare& orig);

    double Square() override;
    void   Print() override;

    virtual ~Foursquare();
private:
    size_t side;
};

#endif      /* FOURSQUARE_H */
```

- Foursquare.cpp

```
#include <cstdlib>
#include <iostream>
#include "Foursquare.h"


Foursquare::Foursquare() : Foursquare(0) {
}

Foursquare::Foursquare(size_t i) : side(i) {
    std::cout << "Foursquare created: " << side << std::endl;
}

Foursquare::Foursquare(std::istream &is) {
    is >> side;
    if (!is){
        side=0;
        is.clear();
        is.ignore();
    }
}

Foursquare::Foursquare(const Foursquare& orig) {
    std::cout << "Foursquare copy created" << std::endl;
```

```
    side = orig.side;
}

double Foursquare::Square() {
    return side*side;

}

void Foursquare::Print() {
    std::cout << "a=" << side  << std::endl;

}

Foursquare::~Foursquare() {
    std::cout << "Foursquare deleted" << std::endl;
}
```

- Octahedron.h

```
#ifndef OCTAHEDRON_H
#define     OCTAHEDRON_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Octahedron : public Figure{
public:
    Octahedron();
    Octahedron(std::istream &is);
    Octahedron(size_t i);
    Octahedron(const Octahedron& orig);

    double Square() override;
    void   Print() override;

    virtual ~Octahedron();
private:
    size_t side;
};

#endif      /* OCTAHEDRON_H */
```

- Octahedron.cpp

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include "Octahedron.h"


Octahedron::Octahedron() : Octahedron(0) {
}

Octahedron::Octahedron(size_t i) : side(i) {
    std::cout << "Octahedron created: " << side << std::endl;
}

Octahedron::Octahedron(std::istream &is) {
    is >> side;
     if (!is){
        side=0;
        is.clear();
        is.ignore();
```

```
    }
}

Octahedron::Octahedron(const Octahedron& orig) {
    std::cout << "Octahedron copy created" << std::endl;
    side = orig.side;
}

double Octahedron::Square() {
    return 2*side*side*(1+sqrt(2));

}

void Octahedron::Print() {
    std::cout << "a=" << side  << std::endl;

}

Octahedron::~Octahedron() {
    std::cout << "Octahedron deleted" << std::endl;
}
```

- Triangle.h

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Triangle : public Figure{
public:
    Triangle();
    Triangle(std::istream &is);
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    double Square() override;
    void   Print() override;

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

- Triangle.cpp

```
    #include <cstdlib>
#include <cmath>
#include <iostream>
#include "Triangle.h"
Triangle::Triangle() : Triangle(0, 0, 0) {
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}
```

```
Triangle::Triangle(std::istream &is) {
    is >> side_a;
    if (!is){
        side_a=0;
        is.clear();
        is.ignore();
    }
    is >> side_b;
    if (!is){
        side_b=0;
        is.clear();
        is.ignore();
    }
    is >> side_c;
    if (!is){
        side_c=0;
        is.clear();
        is.ignore();
    }
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

double Triangle::Square() {
    double p = double(side_a + side_b + side_c) / 2.0;
    return sqrt(p * (p - double(side_a))*(p - double(side_b))*(p -
double(side_c)));
}
void Triangle::Print() {
    std::cout << "a=" << side_a << ", b=" << side_b << ", c=" << side_c <<
std::endl;

}
Triangle::~Triangle() {
    std::cout << "Triangle deleted" << std::endl;
}
```

- main.cpp

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include "Triangle.h"
#include "Octahedron.h"
#include "Foursquare.h"
int main(int argc, char** argv) {
      char num='0';
      Figure *ptr;
      while(true){
            std::cout << std::endl << "Menu: " << std::endl;
            std::cout << "1) Triangle" << std::endl;
            std::cout << "2) Octahedron" << std::endl;
            std::cout << "3) Square" << std::endl;
            std::cout << "4) Exit" << std::endl;
            std::cin >> num;
            if(num=='4')
                  break;
```

```
            switch(num){
                  case '1':{
                        std::cout <<"Enter sides of Triangle"<< std::endl;
                        ptr = new Triangle(std::cin);
                        std::cout << "Triangle: ";
                        ptr->Print();
                        std::cout << "Square of Triangle: ";
                        std::cout << ptr->Square() << std::endl;
                        delete ptr;
                        break;
                  }
                  case '2':{
                        std::cout <<"Enter side of Octahedron"<< std::endl;
                        ptr = new Octahedron(std::cin);
                        std::cout << "Octahedron: ";
                        ptr->Print();
                        std::cout << "Square of Octahedron: ";
                        std::cout << ptr->Square() << std::endl;
                        delete ptr;
                        break;
                  }
                  case '3':{
                        std::cout <<"Enter side of Square"<< std::endl;
                        ptr = new Foursquare(std::cin);
                        std::cout << "Square: ";
                        ptr->Print();
                        std::cout << "Square of Square: ";
                        std::cout << ptr->Square() << std::endl;
                        delete ptr;
                        break;
                  }
                  default:{
                        std::cout << "Unknown option" << std::endl;;
                        break;
                  }
            }
      }
      return 0;
}
```

Выводы: в данной лабораторной работе я получил навыки программирования классов на языке C++, познакомился с перегрузкой операторов и дружественными функциями.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/1

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №2

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

- Закрепление навыков работы с классами.
- Создание простых динамических структур данных.
- Работа с объектами, передаваемыми «по значению».

Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру (колонка фигура 1), согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<). Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
- Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>). Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
- Классы фигур должны иметь операторы копирования (=).
- Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
- Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).
- Класс-контейнер должен иметь метод по добавлению фигуры в контейнер
- Класс-контейнер должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

Теория

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных в процессе существования в памяти могут изменять

не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый *динамическим распределением памяти*, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

- она не имеет имени; (чаще всего к ней обращаются через указатель на адрес)
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

При **передаче по значению** содержимое аргумента копируется в формальный параметр подпрограммы. Изменения, сделанные в параметре, не влияют на значение переменной, используемой при вызове.

Листинг

- Octahedron.h

```
#ifndef OCTAHEDRON_H
#define OCTAHEDRON_H

#include <iostream>
#include "Figure.h"

class Octahedron: public Figure {
public:
    Octahedron();
    Octahedron(size_t i);
    Octahedron(std::istream &is);

    double Square() override;
    void Print() override;

    Octahedron& operator=(const Octahedron& right);
    bool operator==(const Octahedron& right);

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

    ~Octahedron();

private:
    size_t side;
};

#endif
```

- Octahedron.cpp

```
#include "Octahedron.h"
```

```
#include <cmath>

Octahedron::Octahedron() :
        Octahedron(0) {
}

Octahedron::Octahedron(size_t i) :
        side(i) {
    std::cout << "Octahedron created: " << side << std::endl;
}

Octahedron::Octahedron(std::istream &is) {
    is >> side;
    if (!is) {
        side = 0;
        is.clear();
        is.ignore();
    }
}

double Octahedron::Square() {
    return 2 * side * side * (1 + sqrt(2));

}

void Octahedron::Print() {
    std::cout << "a=" << side << std::endl;

}

Octahedron& Octahedron::operator =(const Octahedron& right) {
    if (this == &right)
        return *this;

    side = right.side;
    return *this;
}

bool Octahedron::operator==(const Octahedron& right) {
    return side == right.side;
}

std::ostream& operator <<(std::ostream& os, const Octahedron& octahedron) {
    os << "Octahedron: ";
    os << "a=" << octahedron.side << std::endl;
    return os;
}

std::istream& operator >>(std::istream& is, Octahedron& octahedron) {
    std::cout << "Enter sides of octahedron: ";
    is >> octahedron.side;
    if (!is) {
        octahedron.side = 0;
        is.clear();
        is.ignore();
    }
    return is;
}
Octahedron::~Octahedron() {
    std::cout << "Octahedron deleted" << std::endl;

}
```

- TList.h

```
#ifndef TLIST_H
#define TLIST_H
#include "Octahedron.h"

struct Item {
    Octahedron data;
    Item* next;
};

class TList {
private:
    Item *head;
    int size;
public:

    TList();
    bool Empty();
    int Size();
    Octahedron Fetch(const int index);
    void Insert(const int index, const Octahedron value);
    void PushBack(const Octahedron value);
    void Delete(int ind);

    friend std::ostream& operator <<(std::ostream& os, const TList& array);

    virtual ~TList();
};
#endif
```

- TList.cpp

```
#include "TList.h"

TList::TList() {
    head = nullptr;
    size = 0;
}

bool TList::Empty() {
    return size == 0;
}

int TList::Size() {
    return size;
}

Octahedron TList::Fetch(const int index) {
    if (index < 0 || index > size - 1) {
        std::cerr << "Invalid index" << std::endl;
        return head->data;
    }

    Item* current = head;
    for (int i = 0; i < index; ++i) {
        current = current->next;
    };
    return current->data;
}

void TList::Insert(const int index, const Octahedron value) {
    //idea
    //find pair and insert element between them
    //element stands on position of second one from pair
```

```
    //exceptional situations with no pair:
    //element last or first
    if (index == size) {
        //if element is last
        //no need to shift anything
        PushBack(value);
        return;
    }
    else if (index < 0 || index > size) {
        std::cerr << "Invalid index" << std::endl;
        return;
    }
    else {
        if (index == 0) {
            //element become new head
            Item* shifted = head;
            head = new Item;
            head->next = shifted;
            head->data = value;
            size++;
            return;
        }
        Item* current = head;
        for (int i = 1; i < index; ++i) {
            current = current->next;
        };
        Item* prev = current;
        Item* shifted = prev->next;

        current = new Item;
        prev->next = current;
        current->next = shifted;
        current->data = value;
        size++;
    }
    return;
}

void TList::PushBack(const Octahedron value) {

    Item* item = new Item;

    Item* current = head;
    //running thru all list
    for (int i = 1; i < size; ++i) {
        current = current->next;
    }
    //if list empty
    //item is new head
    if (size == 0) {
        current = item;
        head = current;
    }
    else {
        //stands after previous last element
        current->next = item;
    }
    //ext to last is head
    //for cyclic list
    item->next = head;
    item->data = value;
    size++;
    return;
}
```

```
void TList::Delete(int ind) {
    if (ind < 0 || ind > size - 1) {
        std::cerr << "Invalid index" << std::endl;
        return;
    }
    else {
        if (ind == 0) {
            Item* second = head->next;
            delete head;
            head = second;
            size--;
            return;
        }
        Item* current = head;
        for (int i = 1; i < ind; ++i) {
            current = current->next;
        }
        Item* aftercur = current->next->next;
        delete current->next;
        current->next = aftercur;
        size--;
    }
    return;
}

std::ostream& operator <<(std::ostream& os, const TList& list) {
    if (list.size == 0)
        os << "Empty" << std::endl;
    Item* current = list.head;
    for (int i = 0; i < list.size; ++i) {
        os << current->data;
        current = current->next;
    }
    return os;
}

TList::~TList() {
    for (int i = 0; i < size; ++i)
        Delete(0);
    size = 0;
}
```

- main.cpp

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

int main() {
    TList* list = new TList();

    unsigned int action;

    while (true) {
        std::cout << "Menu:" << std::endl;
        std::cout << "1) Add figure in the end" << std::endl;
        std::cout << "2) Add figure at index" << std::endl;
        std::cout << "3) Delete figure at index" << std::endl;
        std::cout << "4) Fetch figure at index" << std::endl;
        std::cout << "5) Print" << std::endl;
        std::cout << "6) Print size" << std::endl;
        std::cout << "0) Quit" << std::endl;
        std::cin >> action;
```

```
        if (action == 0) {
            break;
        }

        switch (action) {
        case 1: {
            Octahedron t;
            std::cin >> t;
            list->PushBack(t);
            break;
        }
        case 2: {
            int dIndex;
            std::cout << "Enter index: ";
            std::cin >> dIndex;
            Octahedron t;
            std::cin >> t;
            list->Insert(dIndex, t);
            break;
        }

        case 3: {
            int dIndex;
            std::cout << "Enter index: ";
            std::cin >> dIndex;
            list->Delete(dIndex);

            break;
        }

        case 4: {
            int dIndex;
            std::cout << "Enter index: ";
            std::cin >> dIndex;
            std::cout << list->Fetch(dIndex);
            break;
        }
        case 5: {
            std::cout << (*list);
            break;
        }
        case 6: {
            std::cout << list->Size() << std::endl;
            break;
        }

        default:
            std::cout << "Invalid action" << std::endl;
            break;
        }
    }

    delete list;
    return 0;
}
```

Выводы: в данной лабораторной работе я закрепил навыки программирования классов на языке C++, изучил и применил динамические структуры.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/2

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №3

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

- Закрепление навыков работы с классами.
- Знакомство с умными указателями.

Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Класс-контейнер должен соджержать объекты используя std:shared_ptr<...>.
- Класс-контейнер должен иметь метод по добавлению фигуры в контейнер
- Класс-контейнер должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).
- Программа должна позволять:
- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

Теория

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).

Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

В новом стандарте появились следующие умные указатели: *unique_ptr*, *shared_ptr* и *weak_ptr*. Все они объявлены в заголовочном файле `<memory>`.

unique_ptr

Этот указатель пришел на смену старому и проблематичному `auto_ptr`. Основная проблема последнего заключается в правах владения. Объект этого класса теряет права владения ресурсом при копировании (присваивании, использовании в конструкторе копий, передаче в функцию по значению).

shared_ptr

Это самый популярный и самый широкоиспользуемый умный указатель. Он начал своё развитие как часть библиотеки boost. Данный указатель был столь успешным, что его включили в C++ Technical Report 1 и он был доступен в пространстве имен `tr1` — `std::tr1::shared_ptr<>`.

В отличии от рассмотренных выше указателей, `shared_ptr` реализует подсчет

ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0. Как видно, система реализует одно из основных правил сборщика мусора.

weak_ptr

Этот указатель также, как и `shared_ptr` начал свое рождение в проекте boost, затем был включен в C++ Technical Report 1 и, наконец, пришел в новый стандарт. Данный класс позволяет избежать циклической зависимости, которая может образоваться при использовании `shared_ptr`.

Листинг

- `TList.h`

```
#ifndef TLIST_H
#define TLIST_H
#include "TListItem.h"
#include <memory>
class TList {
private:
    std::shared_ptr<TListItem> head;
    int size;
public:

    TList();
    bool Empty()const;
    int Size()const;
    std::shared_ptr<Figure> Fetch(int index)const;
    void Insert(int index, std::shared_ptr<Figure>& obj);
    void Delete(int ind);

    friend std::ostream& operator <<(std::ostream& os, const TList& array);

    virtual ~TList();
};
#endif
```

- `TList.cpp`

```
#include "TList.h"
#include "TListItem.h"
#include <memory>
#include <iostream>
TList::TList() :
        head(nullptr), size(0) {
}

bool TList::Empty() const {
    return size == 0;
}

int TList::Size() const {
    return size;
}

std::shared_ptr<Figure> TList::Fetch(int index) const {
    if (index < 0 || index > size - 1) {
        std::cerr << "Invalid index" << std::endl;
        return nullptr;
    }
    std::shared_ptr<TListItem> current(head);
    for (int i = 0; i < index; ++i) {
        current = current->GetNext();
    };
    return current->GetValue();
```

```
}

void TList::Insert(int index, std::shared_ptr<Figure> &obj) {
    //idea
    //find pair and insert element between them
    //element stands on position of second one from pair
    //exceptional situations with no pair:
    //element last or first
    std::shared_ptr<TListItem> item(new TListItem(obj));

    if (index < 0 || index > size) {
        std::cerr << "Invalid index" << std::endl;
        return;
    } else if (index == 0) {
        item->SetNext(head);
        this->head = item;
    } else {
        std::shared_ptr<TListItem> prev(head);
        for (int i = 0; i < index - 1; ++i)
            prev = prev->GetNext();

        std::shared_ptr<TListItem> shifted(prev->GetNext());
        prev->SetNext(item);
        item->SetNext(shifted);
    }
    size++;
}

void TList::Delete(int ind) {
    if (ind < 0 || ind > size - 1) {
        std::cerr << "Invalid index" << std::endl;
        return;
    } else if (ind == 0) {
        head = head->GetNext();
    } else {
        std::shared_ptr<TListItem> current = head;
        for (int i = 0; i < ind - 1; ++i)
            current = current->GetNext();
        std::shared_ptr<TListItem> aftercur = current->GetNext()->GetNext();
        current->SetNext(aftercur);
    }
    size--;
    return;
}

std::ostream& operator <<(std::ostream& os, const TList& list) {
    if (list.size == 0)
        os << "Empty" << std::endl;
    for (int i = 0; i < list.Size(); ++i) {
        os << *(list.Fetch(i));
    }
    return os;
}

TList::~TList() {
    for (int i = 0; i < size; ++i)
        Delete(0);
    size = 0;
}
```

- TlistItem.h

```
#ifndef TLISTITEM_H_
#define TLISTITEM_H_
```

```
#include <memory>

#include "Figure.h"

class TListItem {
private:
    std::shared_ptr<TListItem> next;
    std::shared_ptr<Figure> figure;
public:

    TListItem();
    TListItem(const std::shared_ptr<Figure> &);
    std::shared_ptr<Figure> GetValue();
    std::shared_ptr<TListItem> GetNext();
    void SetNext(const std::shared_ptr<TListItem>& obj);
    friend std::ostream& operator <<(std::ostream& os, TListItem& obj);

    virtual ~TListItem();
};

#endif /* TLISTITEM_H_ */
```

- TListItem.cpp

```
#include "TListItem.h"
#include <memory>
TListItem::TListItem() {
    this->figure = nullptr;
}
TListItem::TListItem(const std::shared_ptr<Figure> &figure) {
    this->figure = figure;
}


void TListItem::SetNext(const std::shared_ptr<TListItem>& obj) {
    this->next = obj;
    return;
}

std::shared_ptr<Figure> TListItem::GetValue() {
    return this->figure;
}

std::shared_ptr<TListItem> TListItem::GetNext() {
    return this->next;
}

std::ostream& operator <<(std::ostream& os, TListItem& item) {
    if(item.GetValue()!=nullptr)
        os<<*(item.GetValue());
    return os;
}
TlistItem::~TListItem(){}
```

- main.c

```
#include <iostream>
#include "TList.h"
#include "Triangle.h"
#include "Octahedron.h"
#include "Foursquare.h"
int main() {
    TList* list = new TList();
    std::shared_ptr<Figure> fig;
```

```
    unsigned int action;

    while (true) {
        std::cout << "Menu:" << std::endl;
        std::cout << "1) Add figure at index" << std::endl;
        std::cout << "2) Delete figure at index" << std::endl;
        std::cout << "3) Fetch figure at index" << std::endl;
        std::cout << "4) Print" << std::endl;
        std::cout << "5) Print size" << std::endl;
        std::cout << "0) Quit" << std::endl;
        std::cin >> action;

        if (action == 0) {
            break;
        }

        switch (action) {
        case 1: {
            char figure_type;
            std::cout
                    << "Enter figure type: t - triangle, s - foursquare, o -
octahedron: ";
            std::cin >> figure_type;

            int dIndex;
            std::cout << "Enter index: ";
            std::cin >> dIndex;
            switch (figure_type) {
            case 't': {
                std::cout << "Enter sides of triangle" << std::endl;
                fig = std::make_shared<Triangle>(std::cin);
                list->Insert(dIndex,fig);
                break;
            }
            case 's': {

                std::cout << "Enter side of square" << std::endl;
                fig = std::make_shared<Foursquare>(std::cin);
                list->Insert(dIndex,fig);
                break;
            }
            case 'o': {

                std::cout << "Enter side of octahedron" << std::endl;
                fig = std::make_shared<Octahedron>(std::cin);
                list->Insert(dIndex,fig);
                break;
            }
            default: {
                std::cout << "Try more..." << std::endl;
                break;
            }
            }

            break;
        }

        case 2: {
            int dIndex;
            std::cout << "Enter index: ";
            std::cin >> dIndex;
            list->Delete(dIndex);

            break;
```

```
        }

        case 3: {
            int dIndex;
            std::cout << "Enter index: ";
            std::cin >> dIndex;
            std::cout << *(list->Fetch(dIndex));
            break;
        }
        case 4: {
            std::cout << (*list);
            break;
        }
        case 5: {
            std::cout << list->Size() << std::endl;
            break;
        }

        default:
            std::cout << "Invalid action" << std::endl;
            break;
        }
    }

    delete list;
    return 0;
}
```

Выводы: в данной лабораторной работе я закрепил навыки программирования классов на языке C++, изучил и применил умные указатели. Реализовал контейнер хранящий три различные фигуры по ссылке.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/3

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №4

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

• Знакомство с шаблонами классов.

• Построение шаблонов динамических структур данных.

Задание

Необходимо спроектировать и запрограммировать на языке C++ шаблон класса-контейнера первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.

• Шаблон класса-контейнера должен соджержать объекты используя std:shared_ptr<...>.

• Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.

• Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (опередеряется структурой контейнера).

• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опередеряется структурой контейнера).

• Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Теория

Шаблоны — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

В C++ возможно создание шаблонов функций и классов.

Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем, ссылка).

Любой шаблон начинается со слова `template`, будь то шаблон функции или шаблон класса. После ключевого слова `template` идут угловые скобки — `< >`, в которых перечисляется список параметров шаблона. Каждому параметру должно предшествовать зарезервированное слово `class` или `typename`. Отсутствие этих ключевых слов будет расцениваться компилятором как синтаксическая ошибка.

Ключевое слово `typename` говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как: `int`, `double`,`float`, `char` и т. д. А ключевое слово `class` сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться пользовательские типы данных, то есть классы. Но не в коем случае не путайте параметр шаблона и шаблон класса.

Листинг

- `Tlist.h`

```
#ifndef TLIST_H
#define TLIST_H

#include <memory>
#include <iostream>
```

```
#include "TListItem.h"
#include "TListItem.cpp"

template<class T>
class TList {
private:
	std::shared_ptr<TListItem<T>> head;
	int size;
public:
	TList();
	bool Empty() const;
	int Size() const;
	std::shared_ptr<T> Fetch(int index) const;
	void Insert(int index, std::shared_ptr<T>& obj);
	void Delete(const int ind);

	template<class TT>
	friend std::ostream& operator <<(std::ostream& os,
			const TList<TT>& list);

	virtual ~TList();
};
#endif
```

- `Tlist.cpp`

```
#include "TList.h"
template<class T>
TList<T>::TList() :
		head(nullptr), size(0) {
}

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

template<class T>
int TList<T>::Size() const {
	return size;
}

template<class T>
std::shared_ptr<T> TList<T>::Fetch(int index)const {
	if (index < 0 || index > size - 1) {
		std::cerr << "Invalid index" << std::endl;
		return head->GetValue();
	}
	std::shared_ptr<TListItem<T>> current(head);
	for (int i = 0; i < index; ++i) {
		current = current->GetNext();
	};
	return current->GetValue();
}

template<class T>
void TList<T>::Insert(int index, std::shared_ptr<T>& obj) {
```

```
	//idea
	//find pair and insert element between them
	//element stands on position of second one from pair
	//exceptional situations with no pair:
	//element last or first
	std::shared_ptr<TListItem<T>> item(new TListItem<T>(obj));

	if (index < 0 || index > size) {
		std::cerr << "Invalid index" << std::endl;
		return;
	} else if (index == 0) {
		item->SetNext(head);
		this->head = item;
	} else {
		std::shared_ptr<TListItem<T>> prev(head);
		for (int i = 0; i < index - 1; ++i)
			prev = prev->GetNext();

		std::shared_ptr<TListItem<T>> shifted(prev->GetNext());
		prev->SetNext(item);
		item->SetNext(shifted);
	}
	size++;
	return;
}


template<class T>
void TList<T>::Delete(int ind) {
	if (ind < 0 || ind > size - 1) {
		std::cerr << "Invalid index" << std::endl;
		return;
	} else if (ind == 0) {
		head = head->GetNext();
	} else {
		std::shared_ptr<TListItem<T>> current = head;
		for (int i = 0; i < ind - 1; ++i)
			current = current->GetNext();
		std::shared_ptr<TListItem<T>> aftercur = current->GetNext()->GetNext();
		current->SetNext(aftercur);
	}
	size--;
	return;
}


template<class T>
std::ostream& operator <<(std::ostream& os, const TList<T>& list) {
	if (list.size == 0)
		os << "Empty" << std::endl;
	for (int i = 0; i < list.size; ++i) {
		os << *list.Fetch(i);
```

```
	}
	return os;
}

template<class T>
TList<T>::~TList() {
	for (int i = 0; i < size; ++i)
		Delete(0);
	size = 0;
}
```

- TlistItem.h

```
#ifndef TLISTITEM_H_
#define TLISTITEM_H_

#include <memory>
#include <iostream>

template<class T> class TList;
template<class T>
class TListItem {
private:
	std::shared_ptr<TListItem> next;
	std::shared_ptr<T> value;

	void SetNext(const std::shared_ptr<TListItem> obj);
public:
	friend class TList<T> ;
	TListItem();
	TListItem(std::shared_ptr<T>);
	std::shared_ptr<TListItem> GetNext() const;
	std::shared_ptr<T> GetValue() const;

	template<class TT>
	friend std::ostream& operator <<(std::ostream& os, TListItem<TT>& obj);

	virtual ~TListItem();
};
```

```
#endif /* TLISTITEM_H_ */
```

- TlistItem.cpp

```
#include "TListItem.h"

template<class T>
TListItem<T>::TListItem() {
	value = nullptr;
	next = nullptr;

}
template<class T>
TListItem<T>::TListItem(const std::shared_ptr<T> fig) {
	value = fig;
	next = nullptr;
}

template<class T>
void TListItem<T>::SetNext(const std::shared_ptr<TListItem<T>> fig) {
	next = fig;
	return;
}
template<class T>
std::shared_ptr<TListItem<T>> TListItem<T>::GetNext() const {
	return next;
}
template<class T>
std::shared_ptr<T> TListItem<T>::GetValue() const {
	return value;
}
template<class T>
std::ostream& operator <<(std::ostream& os, TListItem<T>& item) {
	if (item.value)
		os << *(item.value);
	return os;
```

```
}
template<class T>
TListItem<T>::~TListItem() {
}
```

Выводы: в данной лабораторной работе я получил навыки построения шаблонов динамических структур данных. Реализовал шаблонный контейнер.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/3

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №5

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

- Закрепление навыков работы с шаблонами классов.
- Построение итераторов для динамических структур данных.

Задание

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№4) спроектировать и разработать Итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for. Например: for(auto i : stack) std::cout << *i << std::endl;

Теория

Итератор (от англ. iterator — перечислитель) — интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера) и навигацию по ним.

Язык C++ широко использует итераторы в STL, поддерживающей несколько различных типов итераторов, включая 'однонаправленные итераторы', 'двунаправленные итераторы' и 'итераторы произвольного доступа'. Все стандартные шаблоны типов контейнеров реализуют разнообразный, но постоянный набор типов итераторов. Синтаксис стандартных итераторов сделан похожим на обычные указатели языка Си, где операторы `*` и `->` используются для указания элемента, на который указывает итератор, а такие арифметические операторы указателя, как `++`, используются для перехода итератора к следующему элементу.

Итераторы обычно используются парами, один из которых используется для указания текущей итерации, а второй служит для обозначения конца коллекции. Итераторы создаются при помощи соответствующих классов контейнеров, используя такие стандартные методы как `begin()` и `end()`. Функция `begin()` возвращает указатель на первый элемент, а `end()` — на воображаемый несуществующий элемент, следующий за последним.

Листинг

- TIterator.h

```
#ifndef TITERATOR_H_
#define TITERATOR_H_

#include <memory>
#include <iostream>
template<class T>
class TIterator {
public:
	TIterator(std::shared_ptr<T> n) {
		node_ptr = n;
	}

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

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

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

	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<T> node_ptr;
};

#endif /* TITERATOR_H_ */
```

- TList.h

```
#ifndef TLIST_H
#define TLIST_H

#include <memory>
#include <iostream>

#include "TListItem.h"
#include "TListItem.cpp"
#include "TIterator.h"

template<class T>
class TList {
private:
	std::shared_ptr<TListItem<T>> head;
	int size;
public:
	TList();
	bool Empty() const;
	int Size() const;
	std::shared_ptr<T> Fetch(int index) const;
	void Insert(int index, std::shared_ptr<T>& obj);
	void Delete(const int ind);

	TIterator<TListItem<T>> begin();
	TIterator<TListItem<T>> end();

	template<class TT>
	friend std::ostream& operator <<(std::ostream& os,
			const TList<TT>& list);

	virtual ~TList();
};
#endif
```

- TList.cpp

```
template<class T>
TIterator<TListItem<T>> TList<T>::begin() {
```

```
	return TIterator<TListItem<T>>(head);
}

template<class T>
TIterator<TListItem<T>> TList<T>::end() {
	return TIterator<TListItem<T>>(nullptr);
}
```

Выводы: в данной лабораторной работе я получил навыки программирования итераторов на языке C++, закрепил навык работы с шаблонами классов. Добавил с контейнеру возможность навигации при помощи итератора.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/5

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №6

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

- Закрепление навыков по работе с памятью в C++.
- Создание аллокаторов памяти для динамических структур данных.

Задание

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛРNo5) 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 варианта задания).

Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.

Теория

Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. std::allocator -- пример реализации аллокатора из стандартной библиотеки, просто использует new и delete, которые обычно обращаются к системным вызовам malloc и free.

Программист обладает преимуществом над стандартным аллокатором, он знает какое количество памяти будет выделятся чаще, как она будет связана.

Хорошим способом оптимизации программы будет уменьшение количества системных вызовов, которые происходят при аллокации. Аллоцировав сразу большой отрезок памяти и распределяя его, можно добиться положительных эффектов.

Листинг

- TAllocationBlock.h

```
#ifndef TALLOCATIONBLOCK_H
#define TALLOCATIONBLOCK_H
#include <cstdlib>
#include "TLista.h"
#include "TLista.cpp"
class TAllocationBlock {
public:
      TAllocationBlock(size_t size, size_t count);
      virtual ~TAllocationBlock();
      void *allocate();
      void deallocate(void *pointer);
      bool hasFreeBlocks();
      int freeCount();
private:
      size_t _size;      //sizeof blocks in bytes
      int _count; //quantity of blocks
```

```
	char*_usedBlocks;
	TLista<void*> _freeBlocks;

};
#endif
/* TALLOCATIONBLOCK_H */
```

- TAllocationBlock.cpp

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

int TAllocationBlock::freeCount() {
	return _freeBlocks.Size();
}
TAllocationBlock::TAllocationBlock(size_t size, size_t count) :
		_size(size), _count(count) {
	_usedBlocks = (char*) malloc(_size * _count);
	//_free_blocks = (void**) malloc(sizeof(void*) * _count);
	for (int i = 0; i < _count; i++)
		_freeBlocks.Insert(i, (void*) (_usedBlocks + i * _size));
	std::cout << "//TAllocationBlock: Memory init" << std::endl;
}
void *TAllocationBlock::allocate() {
	void *result = nullptr;
	if (freeCount() > 0) {
		result = _freeBlocks.Fetch(freeCount()-1);
		_freeBlocks.Delete(freeCount()-1);
		std::cout << "//TAllocationBlock: Allocate " << (_count -
freeCount())
			<< " of " << _count << std::endl;
	} else {
		std::cout << "//TAllocationBlock: No memory exception" << std::endl;
		result = nullptr;
	}
	return result;
}
void TAllocationBlock::deallocate(void *pointer) {
	std::cout << "//TAllocationBlock: Deallocate block " << std::endl;
	_freeBlocks.Insert(freeCount(), pointer);
}

bool TAllocationBlock::hasFreeBlocks() {
	return freeCount() > 0;
}
TAllocationBlock::~TAllocationBlock() {
	if (freeCount() < _count)
		std::cout << "//TAllocationBlock: Memory leaked" << std::endl;
	else
		std::cout << "//TAllocationBlock: Memory freed" << std::endl;
//	_free_blocks will be deleted after exiting this deconstructor
	delete _usedBlocks;
}
```

- TListItem.h

```
#ifndef TLISTITEM_H_
#define TLISTITEM_H_

#include <memory>
#include <iostream>
#include "TAllocationBlock.h"
#include "TAllocationBlock.cpp"
template<class T> class TList;
template<class T>
```

```
class TListItem {
private:
	std::shared_ptr<TListItem> next;
	std::shared_ptr<T> value;
	static TAllocationBlock allocator;

	void SetNext(const std::shared_ptr<TListItem> obj);
public:
	friend class TList<T> ;
	TListItem();
	TListItem(std::shared_ptr<T>);
	std::shared_ptr<TListItem> GetNext() const;
	std::shared_ptr<T> GetValue() const;

	template<class TT>
	friend std::ostream& operator <<(std::ostream& os, TListItem<TT>& obj);

	void * operator new (size_t size);
	void operator delete(void *p);
	virtual ~TListItem();
};

#endif /* TLISTITEM_H_ */
```

- TListItem.cpp

```
#include "TListItem.h"

template<class T>
TListItem<T>::TListItem() {
	value = nullptr;
	next = nullptr;

}
template<class T>
TListItem<T>::TListItem(const std::shared_ptr<T> fig) {
	value = fig;
	next = nullptr;
}

template<class T>
TAllocationBlock TListItem<T>::allocator(
		sizeof(TListItem<T> ), 20);


template<class T>
void * TListItem<T>::operator new(size_t size) {
	std::cout<<"//TListItem: new item"<<std::endl;
	return allocator.allocate();
}
template<class T>
void TListItem<T>::operator delete(void *p) {
	std::cout<<"//TListItem: item deleted"<<std::endl;
	allocator.deallocate(p);
}

template<class T>
void TListItem<T>::SetNext(const std::shared_ptr<TListItem<T>> fig) {
	next = fig;
	return;
}
template<class T>
std::shared_ptr<TListItem<T>> TListItem<T>::GetNext() const {
	return next;
```

```
}
template<class T>
std::shared_ptr<T> TListItem<T>::GetValue() const {
	return value;
}
template<class T>
std::ostream& operator <<(std::ostream& os, TListItem<T>& item) {
	if (item.value)
		os << *(item.value);
	return os;
}
template<class T>
TListItem<T>::~TListItem() {}
```

- TLista.h

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

template<class T>
class TLista {

public:
	struct Item {
		T data;
		Item* next;
	};
	TLista();
	bool Empty() const;
	int Size() const;
	T Fetch(const int index) const;
	void Insert(int index, T obj);
	void Delete(const int ind);


	template<class T2>
	friend std::ostream& operator <<(std::ostream& os, const TLista<T2>&
array);

	virtual ~TLista();
private:
	Item* head;
	int size;
};

#endif //TLISTA_H
```

- TLista.cpp

```
#include "TLista.h"
template<class T>
TLista<T>::TLista() :
		head(nullptr), size(0) {
}

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

template<class T>
int TLista<T>::Size() const {
	return size;
```

```
}

template<class T>
T TLista<T>::Fetch(const int index) const {
	if (index < 0 || index > size - 1) {
		std::cerr << "Invalid index" << std::endl;
		return this->head;
	}
	Item* current = head;
	for (int i = 0; i < index; ++i) {
		current = current->next;
	};
	return current->data;
}

template<class T>
void TLista<T>::Insert(int index, T obj) {
	//idea
	//find pair and insert element between them
	//element stands on position of second one from pair
	//exceptional situations with no pair:
	//element last or first

	if (index < 0 || index > size) {
		std::cerr << "Invalid index" << std::endl;
		return;
	}
	if (index == 0) {
		//element become new head
		Item* shifted = head;
		head = (Item*) malloc(sizeof(Item));
		head->next = shifted;
		head->data = obj;
		size++;
		return;
	}
	Item* prev = head;
	for (int i = 1; i < index; ++i) {
		prev = prev->next;
	};
	Item* shifted = prev->next;
	Item* current = new Item;
	prev->next = current;
	current->next = shifted;
	current->data = obj;
	size++;
	return;
}

template<class T>
void TLista<T>::Delete(int ind) {
	if (ind < 0 || ind > size - 1) {
		std::cerr << "Invalid index" << std::endl;
		return;
	}

	if (ind == 0) {
		Item* item = head->next;
		delete head;
		head = item;
		size--;
		return;
	}
	Item* current = head;
```

```
	for (int i = 1; i < ind; ++i) {
		current = current->next;
	}
	Item*aftercur = current->next->next;
	delete current->next;
	current->next = aftercur;
	size--;

	return;
}

template<class T>
std::ostream& operator <<(std::ostream& os, const TLista<T>& list) {
	if (list.size == 0)
		os << "Empty" << std::endl;
	for (int i = 0; i < list.size; ++i) {
		os << *list.Fetch(i);
	}
	return os;
}

template<class T>
TLista<T>::~TLista() {
	int count = this->Size();
	for (int i = 0; i < count; ++i) {
		std::cout << "//_freeBlocks element deleted: " << Fetch(0) <<
std::endl;
		Delete(0);
	}
	size = 0;
}
```

Выводы: в данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки создания аллокаторов. Добавил аллокатор и упрощенный список, в котором будут хранится адреса использованных/свободных блоков.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/6

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №7

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

• Создание сложных динамических структур данных.

• Закрепление принципа OCP.

Задание

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер, одного из следующих видов (Контейнер 1-го уровня):

1. Массив
2. Связанный список
3. Бинарное Дерево.
4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).
5. Очередь
6. Стек

Каждым элементом контейнера, в свою, является динамической структурой данных одного из следующих видов (Контейнер 2-го уровня):

1. Массив
2. Связанный список
3. Бинарное- Дерево.
4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).
5. Очередь
6. Стек

Таким образом у нас получается контейнер в контейнере. Т.е. для варианта (1,2) это будет массив, каждый из элементов которого – связанный список.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше 5.

Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Например, для варианта (1,2) добавление объектов будет выглядеть следующим образом:

1. Вначале массив пустой.
2. Добавляем Объект1: В массиве по индексу 0 создается элемент с типом список, в список добавляется Объект 1.
3. Добавляем Объект2: Объект добавляется в список, находящийся в массиве по индекс 0.
4. Добавляем Объект3: Объект добавляется в список, находящийся в массиве по индекс 0.
5. Добавляем Объект4: Объект добавляется в список, находящийся в массиве по индекс 0.
6. Добавляем Объект5: Объект добавляется в список, находящийся в массиве по индекс 0.
7. Добавляем Объект6: В массиве по индексу 1 создается элемент с типом список, в список добавляется Объект 6. Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта

Листинг

- `TRemoveCriteria.h`

```
#ifndef LISTS_TREMOVECRITERIA_H_
#define LISTS_TREMOVECRITERIA_H_
#include <memory>
template<class T>
```

```
class TRemoveCriteria {
public:
      virtual bool isIt(std::shared_ptr<T>& obj) = 0;
      virtual ~TRemoveCriteria() {
      }
      ;
private:
};

#endif /* LISTS_TREMOVECRITERIA_H_ */
```

- TRemoveCriteriaAll.h

```
#ifndef LISTS_TREMOVECRITERIAALL_H_
#define LISTS_TREMOVECRITERIAALL_H_

#include "TRemoveCriteria.h"
template<class T>
class TRemoveCriteriaAll: public TRemoveCriteria<T> {
public:
      TRemoveCriteriaAll() {  }
      ;
      bool isIt(std::shared_ptr<T>& obj) override {
            return true;
      }
private:
};

#endif /* LISTS_TREMOVECRITERIAALL_H_ */
```

- TRemoveCriteriaByType.h

```
#ifndef LIST_TREMOVECRITERIABYTYPE_H_
#define LIST_TREMOVECRITERIABYTYPE_H_


#include "TRemoveCriteria.h"
template<class T>
class TRemoveCriteriaByType: public TRemoveCriteria<T> {
public:
      TRemoveCriteriaByType(double value) :
                  _value(value) {
      }
      ;
      bool isIt(std::shared_ptr<T>& obj) override {
            return _value == obj->getType();
      }
private:
      char _value;
};


#endif /* LIST_TREMOVECRITERIABYTYPE_H_ */
```

- TRemoveCriteriaByValue.h

```
#ifndef LISTS_TREMOVECRITERIABYVALUE_H_
#define LISTS_TREMOVECRITERIABYVALUE_H_

#include "TRemoveCriteria.h"

template<class T>
class TRemoveCriteriaByValue: public TRemoveCriteria<T> {
public:
      TRemoveCriteriaByValue(double value) :
                  _value(value) {
```

```
	}
	;
	bool isIt(std::shared_ptr<T>& obj) override {
		return _value > obj->Square();
	}
private:
	double _value;
};
#endif /* LISTS_TREMOVECRITERIABYVALUE_H_ */
```

- TListStorage.h

```
#ifndef TLISTSTORAGE_H_
#define TLISTSTORAGE_H_

#include "TListStorageItem.h"
#include "TListStorageItem.cpp"
#include "TIterator.h"
#include "TList.h"
#include "TList.cpp"
#include "TRemoveCriteria.h"
#include <memory>

template<class T>
class TListStorage {
private:
	std::shared_ptr<TListStorageItem<TList<T>>> head;
	const int maxAtWrapper = 5;
public:

	TListStorage();
	//std::shared_ptr<TListItem<T1, T2, T3>> Fetch(const int index) const;
	void Insert(std::shared_ptr<T> obj);
	void RemoveSubitem(TRemoveCriteria<T>* criteria);
	void Optimize();
	void PushBackWrapper(std::shared_ptr<TListStorageItem<TList<T>>>&& value);
	void RemoveWrapper();
	size_t Size();

	TIterator<TListStorageItem<TList<T>>> begin() const;
	TIterator<TListStorageItem<TList<T>>> end() const;

	template<class TT>
	friend std::ostream& operator <<(std::ostream& os,
			const TListStorage<TT>& list);
	virtual ~TListStorage();
};
#endif /* TLISTSTORAGE_H_ */
```

- TListStorage.cpp

```
#include "TListStorage.h"
template<class T>
TListStorage<T>::TListStorage() :
		head(nullptr) {
}
template<class T>
void TListStorage<T>::RemoveSubitem(TRemoveCriteria<T>* criteria) {
	std::shared_ptr<TList<T>> list = nullptr;
	for (auto item : *this) {
		list = item->GetValue();
		for (int i = 0; i < list->Size(); ++i) {
			std::shared_ptr<T> value = list->Fetch(i);
			if (criteria->isIt(value)) {
				list->Delete(i);
```

```
                        i--;
                  }
            }
      }
      Optimize();
}
template<class T>
void TListStorage<T>::Optimize() {
      if (!head)
            //empty
            return;
      std::shared_ptr<TListStorageItem<TList<T>>> curr(head);
      std::shared_ptr<TListStorageItem<TList<T>>> next(head->GetNext());

      std::shared_ptr<TList<T>> list1 = curr->GetValue();
      std::shared_ptr<TList<T>> list2(nullptr);
      std::shared_ptr<T> fig;

      while (next) {
            list2 = next->GetValue();
            if (list2->Size() > 0)
                  if (list1->Size() < maxAtWrapper) {
                        fig = list2->Fetch(0);
                        list1->Insert(list1->Size(), fig);
                        list2->Delete(0);
                  } else {
                        curr = curr->GetNext();
                        list1 = curr->GetValue();
                        if (curr == next)
                              next=next->GetNext();
                  }
            else
                  next = next->GetNext();
      }
      int emptyCount = 0;
      for (auto item : *this)
            if (item->GetValue()->Size() == 0)
                  emptyCount++;

      for (int i = 0; i < emptyCount; ++i)
            RemoveWrapper();
}

template<class T>
void TListStorage<T>::Insert(std::shared_ptr<T> obj) {

      std::shared_ptr<T> popped = nullptr;
      std::shared_ptr<TList<T>> list = nullptr;
      popped = obj;
      double square = obj->Square();
      bool inserted = false;
      for (auto item : *this) {
            list = item->GetValue();
            if (!inserted) {
                  int i = 0;
                  for (auto elem : *(list))
                        if (elem->GetValue()->Square() >= square) {
                              if (list->Size() == maxAtWrapper) {
                                    popped = list->Fetch(maxAtWrapper - 1);
                                    list->Delete(maxAtWrapper - 1);
                              } else
                                    popped = nullptr;

                              list->Insert(i, obj);
```

```
                              inserted = true;
                              break;
                        } else
                              i++;
                  if (i < maxAtWrapper && !inserted){
                        list->Insert(list->Size(), obj);
                        popped = nullptr;}
            } else {
                  if (list->Size() == maxAtWrapper) {
                        list->Insert(0, popped);
                        popped = list->Fetch(maxAtWrapper);
                        list->Delete(maxAtWrapper);
                  } else {
                        list->Insert(0, popped);
                        popped = nullptr;
                  }
            }
      }
      if (popped != nullptr) {
            PushBackWrapper(
                        std::shared_ptr<TListStorageItem<TList<T>>>(
                                    new TListStorageItem<TList<T>>));

            for (auto item : *this) {

                  list = item->GetValue();
                  if (list->Size() < maxAtWrapper)
                        list->Insert(list->Size(), popped);
            }

      }

}

template<class T>
TIterator<TListStorageItem<TList<T>>> TListStorage<T>::begin() const {
      return TIterator<TListStorageItem<TList<T>>>(this->head);
}

template<class T>
TIterator<TListStorageItem<TList<T>>> TListStorage<T>::end() const {
      return TIterator<TListStorageItem<TList<T>>>(nullptr);
}

template<class T>
void TListStorage<T>::PushBackWrapper(
            std::shared_ptr<TListStorageItem<TList<T>>>&& value) {
      if (!head)
            head = value;
      else {
            head->PushBack(value);
      }
}
template<class T>
void TListStorage<T>::RemoveWrapper() {
      if (!head)
            //empty
            return;
      std::shared_ptr<TListStorageItem<TList<T>>> item = head;
      std::shared_ptr<TListStorageItem<TList<T>>> prev = nullptr;

      while (item->GetNext()) {
            prev = item;
            item = item->GetNext();
```

```
	}
	if (!prev)
		head = nullptr;
	else
		prev->SetNext(nullptr);
}
template<class T>
std::ostream& operator <<(std::ostream& os, const TListStorage<T>& storage) {
	std::shared_ptr<TList<T>> list;
	if (!storage.head)
		os << "Empty" << std::endl;
	for (auto item : storage) {
		list = item->GetValue();
		os << '[' << *list << ']' << std::endl;
	}
	return os;
}
template<class T>
TListStorage<T>::~TListStorage() {
	std::cout << "#Storage deleted" << std::endl;
}
```

- TListStorageItem.h

```
#ifndef TLISTSTORAGEITEM_H_
#define TLISTSTORAGEITEM_H_
#include <memory>

template<class T>
class TListStorageItem {
private:
	std::shared_ptr<TListStorageItem<T>> next;
	std::shared_ptr<T> value;
public:
	TListStorageItem();
	std::shared_ptr<T> GetValue() const;
	std::shared_ptr<TListStorageItem<T>> GetNext() const;
	void SetNext(std::shared_ptr<TListStorageItem<T>> next);
	void PushBack(std::shared_ptr<TListStorageItem<T>> next);
	virtual ~TListStorageItem();
};

#endif /* TLISTSTORAGEITEM_H_ */
```

- TListStorageItem.cpp

```
#include "TListStorageItem.h"

template<class T>
TListStorageItem<T>::TListStorageItem() :
		next(nullptr) {
	value = std::shared_ptr<T>(new T);

}
template<class T>
std::shared_ptr<T> TListStorageItem<T>::GetValue() const {
	return value;
}

template<class T>
void TListStorageItem<T>::PushBack(std::shared_ptr<TListStorageItem> value) {
	if (!next) {
		next = value;
	} else {
		next->PushBack(value);
```

```
	}
}

template<class T>
std::shared_ptr<TListStorageItem<T>> TListStorageItem<T>::GetNext() const {
	return next;
}

template<class T>
void TListStorageItem<T>::SetNext(std::shared_ptr<TListStorageItem<T>> next) {
	this->next = next;
}

template<class T>
TListStorageItem<T>::~TListStorageItem() {
}
;
```

- main.cpp

```
#include <iostream>

#include "list/TListStorage.h"
#include "list/TListStorage.cpp"
#include "list/TRemoveCriteriaAll.h"
#include "list/TRemoveCriteriaByValue.h"
#include "list/TRemoveCriteriaByType.h"
#include "figure/Figure.h"
#include "figure/Foursquare.h"
#include "figure/Octahedron.h"
#include "figure/Triangle.h"

int main(int argc, char** argv) {
	TListStorage<Figure> list;
	unsigned int action;
	while (true) {
		std::cout<<"====================================="<<std::endl;
		std::cout << "Menu:" << std::endl;
		std::cout << "1) Add figure" << std::endl;
		std::cout << "2) Delete figures with square lower" << std::endl;
		std::cout << "3) Delete figures with type" << std::endl;
		std::cout << "4) Delete all figures" << std::endl;
		std::cout << "5) Print" << std::endl;
		std::cout << "0) Quit" << std::endl;
		std::cin >> action;
		if (action == 0) {
			break;
		}
		switch (action) {
		case 1: {
			char figure_type;
			std::cout
					<< "Enter figure type: t - triangle, s -
foursquare, o - octahedron: ";
			std::cin >> figure_type;
			switch (figure_type) {
			case 't': {
				std::cout << "Enter sides of triangle" << std::endl;
				list.Insert(std::shared_ptr<Triangle>(new
Triangle(std::cin)));
				break;
			}
			case 's': {
				std::cout << "Enter side of square" << std::endl;
```

```
                        list.Insert(
                                    std::shared_ptr<Foursquare>(new
Foursquare(std::cin)));
                        break;
                  }
                  case 'o': {
                        std::cout << "Enter side of octahedron" << std::endl;
                        list.Insert(
                                    std::shared_ptr<Octahedron>(new
Octahedron(std::cin)));
                        break;
                  }
                  default: {
                        std::cout << "Try more..." << std::endl;
                        break;
                  }
                  }

                  break;
            }
            case 2: {
                  int square;
                  std::cout << "Enter square: ";
                  std::cin >> square;
                  TRemoveCriteriaByValue<Figure> criteria(square);
                  list.RemoveSubitem(&criteria);
                  break;
            }
            case 3: {
                  char figure_type;
                  std::cout
                              << "Enter figure type: t - triangle, s -
foursquare, o - octahedron: ";

                  std::cin >> figure_type;
                  TRemoveCriteriaByType<Figure> criteria(figure_type);
                  list.RemoveSubitem(&criteria);
                  break;
            }
            case 4: {
                  TRemoveCriteriaAll<Figure> criteria;
                  list.RemoveSubitem(&criteria);
                  break;
            }
            case 5: {
                  std::cout << list << std::endl;
                  break;
            }

            default:
                  std::cout << "Invalid action" << std::endl;
                  break;
            }
      }
      return 0;
}
```

Выводы: в данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки создания сложных динамических. Создал сложно хранилище данных: список со списками, автосортируемый по площади, с возможностью удаления по критериям.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/6

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №8

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

• Знакомство с параллельным программированием в C++.

Задание

Используя структуры данных, разработанные для лабораторной работы No6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера .

Необходимо разработать два вида алгоритма:

• Обычный, без параллельных вызовов.

• С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

• future

• packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

• mutex

• lock_guard

Теория

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций. Теория параллельных вычислений составляет раздел прикладной теории алгоритмов.

Существуют различные способы реализации параллельных вычислений. Например, каждый вычислительный процесс может быть реализован в виде процесса операционной системы, либо же вычислительные процессы могут представлять собой набор потоков выполнения внутри одного процесса ОС. Параллельные программы могут физически исполняться либо последовательно на единственном процессоре — перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно — выделяя каждому вычислительному процессу один или несколько процессоров (находящихся рядом или распределённых в компьютерную сеть).

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один или отсутствуют элементы.

Листинг

- `TList.h`

```
#ifndef TLIST_H
#define TLIST_H

#include <memory>
```

```
#include <iostream>
#include <future>
#include <mutex>
#include <functional>
#include "TListItem.h"
#include "TListItem.cpp"
#include "TIterator.h"

template<class T>
class TList {
private:
	std::shared_ptr<TListItem<T>> head;
	int size;
	std::future<void> SortThreadTask();
public:
	TList();
	bool Empty() const;
	int Size() const;
	std::shared_ptr<T> Fetch(int index) const;
	std::shared_ptr<T> Pop(int index);
	void Insert(int index, std::shared_ptr<T> obj);
	void Delete(const int ind);
	std::shared_ptr<T> operator[](size_t i);
	void Sort();
	void SortParallel();
	TIterator<TListItem<T>> begin();
	TIterator<TListItem<T>> end();

	template<class TT>
	friend std::ostream& operator <<(std::ostream& os, const TList<TT>& list);

	virtual ~TList();
};
#endif
```

- TList.cpp

```
#include "TList.h"
template<class T>
TList<T>::TList() :
		head(nullptr), size(0) {
}

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

template<class T>
int TList<T>::Size() const {
	return size;
}
template<class T>
std::shared_ptr<T> TList<T>::operator[](size_t i) {
	return Fetch(i);
}

template<class T>
void TList<T>::Sort() {
	if (Size() > 1) {
		std::shared_ptr<T> middle = Pop(0);
		TList<T> left, right;
		while (!Empty()) {
			std::shared_ptr<T> item = Pop(0);
```

```
                if (*item > *middle) {
                        left.Insert(0, item);
                } else {
                        right.Insert(0, item);
                }
        }
        left.Sort();
        right.Sort();
        while (!left.Empty())
                Insert(0, left.Pop(left.Size() - 1));
        Insert(0, middle);
        while (!right.Empty())
                Insert(0, right.Pop(right.Size() - 1));
    }
}
template<class T>
void TList<T>::SortParallel() {
    if (Size() > 1) {
        std::shared_ptr<T> middle = Pop(0);
        TList<T> left, right;
        while (!Empty()) {
                std::shared_ptr<T> item = Pop(0);
                if (*item > *middle)
                        left.Insert(0, item);
                else
                        right.Insert(0, item);

        }
        std::future<void> left_res = left.SortThreadTask();
        std::future<void> right_res = right.SortThreadTask();
        left_res.get();
        while (!left.Empty())
                Insert(0, left.Pop(left.Size() - 1));
        Insert(0, middle);
        right_res.get();
        while (!right.Empty())
                Insert(0, right.Pop(right.Size() - 1));
    }
}
template<class T>
std::future<void> TList<T>::SortThreadTask() {
    std::future<void> res = std::async(std::launch::async,
                &TList<T>::SortParallel, this);
    return res;
}
template<class T>
std::shared_ptr<T> TList<T>::Fetch(int index) const {
    if (index < 0 || index > size - 1) {
        std::cerr << "Invalid index" << std::endl;
        return nullptr;
    }
    std::shared_ptr<TListItem<T>> current(head);
    for (int i = 0; i < index; ++i) {
        current = current->GetNext();
    };
    return current->GetValue();
}

template<class T>
void TList<T>::Insert(int index, std::shared_ptr<T> obj) {
    //idea
    //find pair and insert element between them
    //element stands on position of second one from pair
    //exceptional situations with no pair:
```

```
	//element last or first
	std::shared_ptr<TListItem<T>> item(new TListItem<T>(obj));

	if (index < 0 || index > size) {
		std::cerr << "Invalid index" << std::endl;
		return;
	} else if (index == 0) {
		item->SetNext(head);
		this->head = item;
	} else {
		std::shared_ptr<TListItem<T>> prev(head);
		for (int i = 0; i < index - 1; ++i)
			prev = prev->GetNext();

		std::shared_ptr<TListItem<T>> shifted(prev->GetNext());
		prev->SetNext(item);
		item->SetNext(shifted);
	}
	size++;
	return;
}

template<class T>
std::shared_ptr<T> TList<T>::Pop(int index) {
	std::shared_ptr<T> res = Fetch(index);
	if (res)
		Delete(index);
	return res;
}

template<class T>
void TList<T>::Delete(int ind) {
	if (ind < 0 || ind > size - 1) {
		std::cerr << "Invalid index" << std::endl;
		return;
	} else if (ind == 0) {
		head = head->GetNext();
	} else {
		std::shared_ptr<TListItem<T>> current = head;
		for (int i = 0; i < ind - 1; ++i)
			current = current->GetNext();
		std::shared_ptr<TListItem<T>> aftercur = current->GetNext()-
>GetNext();
		current->SetNext(aftercur);
	}
	size--;
	return;
}
template<class T>
TIterator<TListItem<T>> TList<T>::begin() {
	return TIterator<TListItem<T>>(head);
}

template<class T>
TIterator<TListItem<T>> TList<T>::end() {
	return TIterator<TListItem<T>>(nullptr);
}

template<class T>
std::ostream& operator <<(std::ostream& os, const TList<T>& list) {
	if (list.size == 0)
		os << "Empty" << std::endl;
	for (int i = 0; i < list.size; ++i) {
		os << *list.Fetch(i);
```

```
	}
	return os;
}

template<class T>
TList<T>::~TList() {
	for (int i = 0; i < size; ++i)
		Delete(0);
	size = 0;
}
```

Выводы: в данной лабораторной работе я получил алгоритмы работы с параллельным программированием в C++. Реализовал алгоритм быстрой сортировки и распараллелил его.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/6

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование

Лабораторная работа №9

Студент: Лукашкин К.В
Год приёма: 2017
Группа: М8О-204Б
Преподаватель: Поповкин А. В.
Вариант: №8

Москва, 2017

Цель работы

Целью лабораторной работы является:

• Знакомство с лямбда-выражениями

Задание

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) необходимо разработать:

• Контейнер второго уровня с использованием шаблонов.

• Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контейнером 1-го уровня:

• Генерация фигур со случайным значением параметров;

• Печать контейнера на экран;

• Удаление элементов со значением площади меньше определенного числа;

• В контейнер второго уровня поместить цепочку команд.

• Реализовать цикл, который проходит по всем командам в контенере второго уровня и выполняет их, применяя к контейнеру первого уровня.

Для создания потоков использовать механизмы:

• future

• packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

• mutex

• lock_guard

Теория:

Лямбда-выражение в программировании — специальный синтаксис для определения функциональных объектов, заимствованный из λ-исчисления. Применяется как правило для объявления анонимных функций по месту их использования, и обычно допускает замыкание на лексический контекст, в котором это выражение использовано. Используя лямбда-выражения, можно объявлять функции в любом месте кода.

Листинг

- main.c

```
#include <iostream>
#include <future>
#include <functional>
#include <random>
#include <thread>
#include "list/TList.h"
#include "list/TList.cpp"
#include "figure/Figure.h"
#include "figure/Foursquare.h"
#include "figure/Octahedron.h"
#include "figure/Triangle.h"

int main() {
      typedef std::function<void(void)> command;
      TList<Figure> list;
      TList<command> cmd;
      std::shared_ptr<Figure> fig;
```

```
	command cmd_insert_tri = [&]() {
		std::cout << "Command: Create triangles" << std::endl;
		std::default_random_engine generator;
		std::uniform_int_distribution<int> distribution(1, 20);
		int side;
		for (int i = 0; i < 5; i++) {
			side = distribution(generator);
			list.Insert(0,std::make_shared<Triangle>(side, side, side));
		}
	};
	command cmd_insert_sq = [&]() {
		std::cout << "Command: Create foursquares" << std::endl;
		std::default_random_engine generator;
		std::uniform_int_distribution<int> distribution(1, 20);
		int side;
		for (int i = 0; i < 5; i++) {
			side = distribution(generator);
			list.Insert(0,std::make_shared<Foursquare>(side));
		}
	};
	command cmd_insert_oct = [&]() {
		std::cout << "Command: Create octahedrons" << std::endl;
		std::default_random_engine generator;
		std::uniform_int_distribution<int> distribution(1, 20);
		int side;
		for (int i = 0; i < 5; i++) {
			side = distribution(generator);
			list.Insert(0,std::make_shared<Octahedron>(side));
		}
	};

	command cmd_print = [&]() {
		std::cout << "Command: Print list" << std::endl;
		std::cout << list<<std::endl;
	};

	command cmd_delete = [&]() {
		std::cout << "Command: Delete square lower" << std::endl;
		for(int i=0;i<list.Size();++i) {
			if(list.Fetch(i)->Square()<20) {
				list.Delete(i);
				i--;
			}}
	};

	cmd.Insert(0, std::shared_ptr<command>(&cmd_insert_tri, [](command*) {}));
// using custom deleter
	cmd.Insert(0, std::shared_ptr<command>(&cmd_insert_sq, [](command*)
{})); // using custom deleter
	cmd.Insert(0, std::shared_ptr<command>(&cmd_print, [](command*) {})); //
using custom deleter
	cmd.Insert(0, std::shared_ptr<command>(&cmd_delete, [](command*) {})); //
using custom deleter
	cmd.Insert(0, std::shared_ptr<command>(&cmd_print, [](command*) {})); //
using custom deleter

	while (!cmd.Empty()) {
		std::shared_ptr<command> com = cmd.Pop(cmd.Size()-1);
		std::future<void> ft = std::async(*com);
		ft.get();
	}

	return 0;
}
```

Выводы: в данной лабораторной работе я получил навыки работы с лямбда-выражениями. Добавил с помощью лямбда-выражений команды, выполняющие операции с контейнером 1-го уровня: заполнение его случайно сгенерированными фигурами, вывод контейнера на экран, удаление элементов с площадью меньше заданной.

Ссылка на код: https://github.com/memosiki/oop-3sem/tree/master/5