Московский авиационный институт
(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 8

Тема: Асинхронное программирование

Студент: Кудинов Сергей
Преподаватель: Журавлев А.А.
Дата:
Оценка:

Москва, 2019

1. **Постановка задачи**

Разработать приложение, которое будет считывать из стандартного ввода данные фигур, согласно варианту задания, выводить их характеристики на экран и записывать в файл. Фигуры могут задаваться как своими вершинами, так и другими характеристиками

- Осуществлять ввод из стандартного ввода данных фигур, согласно варианту задания
- Программа должна создавать классы, соответствующие введенным данных фигур
- Программа должна содержать внутренний буфер, в котором помещаются фигуры. Для создания буфера допускается использовать стандартные контейнеры STL. Размер буфера задается параметром командной строки
- При наполнении буфера они должны запускаться на асинхронную обработку, после буфер очищается
- Обработка должна производиться в отдельном потоке
- Реализовать два обработчика, один должен выводить данные в консоль, другой – в файл
- Оба обработчика должны обрабатывать каждый буфер
- В программе должно быть только два потока
- В программе должен быть реализован шаблон Publish-Subscribe. Каждый обработчик должен быть реализован как отдельный подписчик.

2. **Репозиторий github**

https://github.com/StormStudioAndroid2/oop_exercise_0 3/

3. **Описание программы**

Реализован базовый абстрактный класс Figure, имеющий чисто виртуальные функции для вычисления площади, центра и ввода/вывода из потоков. В конструкторах классов-наследников Trapeze, Rhombus и Rectangle реализованы проверки на корректность переданных точек, стоит заметить, что точки в конструкторы передаются в любом порядке. Реализован класс Subscriber, и два его наследника – ConsoleSubscriber и FileSubscriber. Все подписчики хранятся в классе Executor. Кроме этого реализован класс Task, хранящий вектор фигур, который нужно обработать. В классе ThreadFunc массив Task последовательно обработывается.

4. Листинг

программы main.cpp

```
#include <iostream>
#include "Subscriber.hpp"

#include "Figure.hpp"
#include "Rhombus.hpp"
#include "Trapeze.hpp"
#include "Rectangle.hpp"


struct ThreadFunc {
public:
    ThreadFunc(const Executor& executor) : executor(executor) {};

    void addTask(std::unique_ptr<Task> task) {
        std::lock_guard<std::mutex> lock(queueMutex);
        tasks.push(std::move(task));
    }

    void startWorking() {
        working = true;
    }

    void stopWorking() {
        working = false;
    }

    bool isWorking() {
        return working;
    }

    std::condition_variable& getCondition1() {
        return condition1;
    }

    std::condition_variable& getCondition2() {
        return condition2;
    }

    std::mutex& getReadMutex() {
        return readMutex;
    }

    void operator()() {
```

```
        while(true) {
            std::unique_lock<std::mutex>
            mainLock(readMutex); while(!working) {
                condition2.wait(mainLock);
            }
            if(!tasks.empty()) {
                {
                    std::lock_guard<std::mutex> lock(queueMutex);
                    std::shared_ptr<Task> currentTask = std::move(tasks.front());
                    tasks.pop();
                    if(currentTask->isExit()) {
                        break;
                    } else {
                         executor.notify(std::move(currentTask));
                    }
                    this->stopWorking();
                    condition1.notify_one();
                }
            }

        }
    }
private:
    Executor executor;
    std::mutex readMutex;
    std::condition_variable condition1;
    std::condition_variable condition2;
    std::mutex queueMutex;
    std::queue<std::shared_ptr<Task>> tasks;
    bool working = false;
};

int main(int argc, char** argv) {
    unsigned bufferSize;
    if(argc != 2) {
        std::cout << "Need args!" << std::endl;
        return -1;
    }
    bufferSize = std::atoi(argv[1]);
    std::vector<std::shared_ptr<Figure>> figures;
    std::string command;
    int command2;

    std::shared_ptr<Subscriber> consolePrint(new ConsoleSubscriber());
    std::shared_ptr<Subscriber> filePrint(new FileSubscriber());

    Executor executor;
    executor.subscribe(consolePrint);
```

```
executor.subscribe(filePrint);

ThreadFunc func(executor);
std::thread thread(std::ref(func));

while(std::cin >> command) {
    if(command == "exit") {
        std::unique_ptr<Task>t(new Task(true, figures));

        func.addTask(std::move(t));
        func.startWorking();
        func.getCondition2().notify_one();
        break;
    } else if(command == "add") {
        std::shared_ptr<Figure> f;
        std::cout << "1 - Rhombus, 2 - Rectangle, 3 - Trapeze" <<
        std::endl; std::cin >> command2;
        try {
            if(command2 == 1) {
                f = std::make_shared<Rhombus>();
                f->scan(std::cin);
            } else if(command2 == 2) {
                f = std::make_shared<Rectangle>();
                f->scan(std::cin);
            } else if(command2 == 3) {
                f = std::make_shared<Rhombus>();
                f->scan(std::cin);
            } else {
                std::cout << "Wrong input" << std::endl;
            }
            figures.push_back(f);
        } catch(std::exception& e) { std::cerr
            << e.what() << std::endl;
        }
        if(figures.size() == bufferSize) {
            std::unique_ptr<Task>t(new Task(false, figures));
            func.addTask(std::move(t));

            func.startWorking();
            func.getCondition2().notify_one();
            std::unique_lock<std::mutex> lock(func.getReadMutex());
            while(func.isWorking()) {
                func.getCondition1().wait(lock);
            }
            figures.resize(0);
        }
    } else {
        std::cout << "Unknown command" << std::endl;
```

```
        }
    }
    thread.join();
    return 0;
}
```

Trapeze.hpp

```
#pragma once

#include <vector>
#include "Figure.hpp"
class Trapeze: public Figure {
    private:
        Point points[4];
    public:
        Trapeze() = default;
        Trapeze(Point p1, Point p2, Point p3, Point
        p4); Trapeze(std::istream& is);

         double getSquare();
        Point getCenter();
         void print(std::ostream& os) const;
         void scan(std::istream &is);
};
```

Trapeze.cpp

```
#include "Trapeze.hpp"


    Trapeze::Trapeze(Point p1, Point p2, Point p3, Point p4) {
      if (IsParallel(p1,p2,p3,p4)
       && !IsParallel(p1,p3,p2,p4))
       { std::swap(p2, p3);

    } else if (!IsParallel(p1,p2,p3,p4)
        && IsParallel(p1,p3,p2,p4)) {

    } else {

        throw std::logic_error("not Trapeze");
    }
    this->points[0] = p1;
    this->points[1] = p2;
    this->points[2] = p3;
    this->points[3] = p4;
    }
    double Trapeze::getSquare() {
```

```
        return (length(this->points[0],this->points[2])+length(this-
>points[1],this->points[3]))*fabs((this->points[0].y-this->points[1].y))*(0.5);
    }
    void Trapeze::print(std::ostream& os) const {
    os << "Trapeze ";
    for (int i = 0; i < 4; ++i) {
        os << this->points[i] << " p" << i+1 <<" ";
    }
        os << std::endl;

    }

void Trapeze::scan(std::istream &is) {
    Point p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Trapeze(p1,p2,p3,p4);
}
Trapeze::Trapeze(std::istream &is) {
    Point p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Trapeze(p1,p2,p3,p4);
}
Point Trapeze::getCenter() {

    Point p;
    p.x = 0;
    p.y = 0;
    for (size_t i = 0; i <4; ++i) {
        p = p+(points[i]/4);
    }
    return p;
}
```

Rhombus.hpp

```
#pragma once

#include <vector>
#include "Figure.hpp"
class Rhombus: public Figure {
    private:
        Point points[4];
    public:
        Rhombus() = default;
        Rhombus(Point p1, Point p2, Point p3, Point
        p4); Rhombus(std::istream& is);

         double getSquare();
```

```
        Point getCenter();
         void print(std::ostream& os) const;
         void scan(std::istream &is);
};
```

Rhombus.cpp

```
#include "Rhombus.hpp"

   Rhombus::Rhombus(Point p1, Point p2, Point p3, Point p4)
     { if (length(p1, p2) == length(p1, p4)
      && length(p3, p4) == length(p2, p3)
      && length(p1, p2) == length(p2, p3)) {

   } else if (length(p1, p4) == length(p1, p3)

         && length(p2, p3) == length(p2, p4)
         && length(p1, p4) == length(p2, p4))
      { std::swap(p2, p3);
   } else if (length(p1, p3) == length(p1, p2)
            && length(p2, p4) == length(p3, p4)
            && length(p1, p2)== length(p2, p4))
      { std::swap(p3, p4);
   } else {
      throw std::logic_error("not rhombus");
   }
   this->points[0] = p1;
      this->points[1] = p2;
   this->points[2] = p3;
   this->points[3] = p4;

   }
   double Rhombus::getSquare() {

         return length(this->points[1],this->points[3])*length(this->points[0],this->points[2])*0.5;
   }
      void Rhombus::print(std::ostream& os) const {
   os << "Rhombus " << std::endl; for (int i = 0; i <
   4; ++i) {
      os << this->points[i] << std::endl;
   }
      os << std::endl;

   }

void Rhombus::scan(std::istream &is) {
   Point p1,p2,p3,p4;
   is >> p1 >> p2 >> p3 >> p4;
   *this = Rhombus(p1,p2,p3,p4);
}
Rhombus::Rhombus(std::istream &is) {
```

```
    Point p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Rhombus(p1,p2,p3,p4);
}
Point Rhombus::getCenter() {

    Point p;
    p.x = 0;
    p.y = 0;
    for (size_t i = 0; i <4; ++i) {
        p = p+(points[i]/4);
    }
    return p;
}
```

Rectangle.hpp

```
#pragma once

#include <vector>
#include "Figure.hpp"
class Rectangle: public Figure {
    private: Point points[4];
    public:
     Rectangle() = default;
        Rectangle(Point p1, Point p2, Point p3, Point p4);
    Rectangle(std::istream& is);

         double getSquare();
        Point getCenter();
         void print(std::ostream& os) const;
         void scan(std::istream &is);
};
```

Rectangle.cpp

```
#include "Rectangle.hpp"

    Rectangle::Rectangle(Point p1, Point p2, Point p3, Point p4) {
       if (IsRectangle(p1,p2,p3,p4)) {

    } else if (IsRectangle(p2, p3, p1, p4)) {
        std::swap(p2, p1); std::swap(p3,p2);
    } else if (IsRectangle(p3, p1, p2, p4)) {
        std::swap(p3, p1); std::swap(p3,p2);
    } else {
        throw std::logic_error("not rectangle");
    }
    this->points[0] = p1;
    this->points[1] = p2;
    this->points[2] = p3;
    this->points[3] = p4;
    }
    double Rectangle::getSquare() {
```

```
      return length(this->points[0],this->points[1])*length(this-
>points[0],this->points[3]);
    }
            void Rectangle::print(std::ostream& os) const {
    os << "Rectangle p1: ";
    for (int i = 0; i < 4; ++i) {
        os << this->points[i] << "p" << i+1 <<" ";
    }
    os << std::endl;
    }

void Rectangle::scan(std::istream &is) {
    Point p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Rectangle(p1,p2,p3,p4);
}
Rectangle::Rectangle(std::istream &is) {
    Point p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Rectangle(p1,p2,p3,p4);
}
Point Rectangle::getCenter()  {

    Point p;
    p.x = 0;
    p.y = 0;
    for (size_t i = 0; i < 4; ++i) {
        p = p+(points[i]/4);
    }
    return p;
}
```

Figure.hpp

```
#pragma once

#include <vector>
#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
struct Point {
   double x = 0;
   double y = 0;
};
std::istream& operator >> (std::istream& is, Point& p);
std::ostream& operator<< (std::ostream& os, const Point& p);
Point operator+(Point left, Point right); Point operator+(Point
left, int right);
Point operator-(Point left, Point right);
Point operator-(Point left, double right);
Point operator/(Point left, double right);

Point operator*(Point left, double right);
```

```
bool IsOrthogonal(Point a, Point b, Point c);
int IsRectangle(Point a, Point b, Point c, Point d);
bool IsParallel(Point a, Point b, Point c, Point d);
double length(Point left, Point right); class Figure
{

   public:
      virtual double getSquare() = 0;
      virtual Point getCenter() = 0;
         virtual ~Figure() = default;

      virtual void print(std::ostream& os) const = 0;
      virtual void scan(std::istream &is) = 0;
};
std::ostream& operator << (std::ostream& os, const Figure&
fig); std::istream& operator >> (std::istream& is, Figure& fig);
```

Figure.cpp

```
#include "Figure.h"

std::ostream& operator << (std::ostream& os, const Figure& fig) {
   fig.print(os);
   return os;
}

std::istream& operator >> (std::istream& is, Figure& fig) {
   fig.scan(is);
   return is;
}
```

Point.cpp

```
#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
#include "Figure.h"
std::istream& operator >> (std::istream& is, Point& p)
 { return is >> p.x >> p.y;
}

std::ostream& operator<< (std::ostream& os, const Point& p)
 { return os << p.x << " " << p.y;
}
Point operator+(Point left, Point right) {
   Point p;
   p.x = left.x+right.x;
```

```
    p.y = left.y+right.y;
        return p;

}
Point operator+(Point left, int right) {
    Point p;
    p.x = left.x+right;
    p.y = left.y+right;
        return p;

}
Point operator-(Point left, Point right) {
    Point p;
    p.x = left.x-right.x;
    p.y = left.y-right.y;
        return p;

}
Point operator-(Point left, double right) {
    Point p;
    p.x = left.x-right;
    p.y = left.y-right;
        return p;

}
Point operator/(Point left, double right) {
    Point p;
    p.x = left.x/right;
    p.y = left.y/right;
        return p;

}
Point operator*(Point left, double right) {
    Point p;
    p.x = left.x*right;
    p.y = left.y*right;
    return p;
}
double length(Point left, Point right) {
    return sqrt((left.x-right.x)*(left.x-right.x)+(left.y-right.y)*(left.y-right.y));
}
bool IsOrthogonal(Point a, Point b, Point c)
{
    return (b.x - a.x) * (b.x - c.x) + (b.y - a.y) * (b.y - c.y) == 0;
}
bool IsParallel(Point a, Point b, Point c, Point d)
{
    Point a1 = a-b;
```

```
    Point a2 = c-d;
    return ((a1.x*a2.x+a1.y*a2.y)/(length(a,b)*length(c,d))<=-1 ||
(a1.x*a2.x+a1.y*a2.y)/(length(a,b)*length(c,d))>=1); }

int IsRectangle(Point a, Point b, Point c, Point d)
{
    return
        IsOrthogonal(a, b, c) &&
        IsOrthogonal(b, c, d) &&
        IsOrthogonal(c, d, a);
}
```

Subscriber.hpp

```
#pragma once

#include <fstream>
#include <memory>
#include <vector>
#include <queue>
#include <map>
#include <thread>
#include <mutex>
#include <condition_variable>

#include "Figure.hpp"


class Task {
public:
    Task(bool type, const std::vector<std::shared_ptr<Figure>>&
    data); bool isExit() const;
    std::vector<std::shared_ptr<Figure>> getData() const;
private:
    bool type;
    std::vector<std::shared_ptr<Figure>> data;
};

struct Subscriber {
public:
    virtual void print(std::shared_ptr<Task> task) const = 0;
    virtual ~Subscriber() = default;
};

struct ConsoleSubscriber : public Subscriber {
public:
    void print(std::shared_ptr<Task> task) const override;
};
```

```
struct FileSubscriber : public Subscriber {
public:
    void print(std::shared_ptr<Task> task) const override;
};


class Executor {
public:
    void subscribe(std::shared_ptr<Subscriber>& s);

    void notify(std::shared_ptr<Task> task);
private:
    std::vector<std::shared_ptr<Subscriber>> subscribers;
};
```

Subscriber.cpp

```
#include "Subscriber.hpp"


    Task::Task(bool type, const std::vector<std::shared_ptr<Figure>>& data) :
type(type), data(data) {};
    bool Task::isExit() const {
        return type;
    }
    std::vector<std::shared_ptr<Figure>> Task::getData() const
        { return data;
    }




    void ConsoleSubscriber::print(std::shared_ptr<Task> task) const {
        for (size_t i = 0; i<task->getData().size(); ++i) {
            task->getData()[i]->print(std::cout);
        }
    }

    void FileSubscriber::print(std::shared_ptr<Task> task) const {
        // auto data = task->getData();
        std::ofstream os(std::to_string(rand() % 1337) + ".txt");

        for (size_t i = 0; i<task->getData().size(); ++i) {
            task->getData()[i]->print(os);
        }
    }
```

```
void Executor::subscribe(std::shared_ptr<Subscriber>& s)
    { subscribers.push_back(s);
}

void Executor::notify(std::shared_ptr<Task> task)
    { for(const auto& subscriber : subscribers) {

        subscriber->print(task);
    }
}
```

5. Вывод

В результате данной работы я научился работать с Cmake, создавать потоки и работать с ними