

Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»
Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 8: «Асинхронное программирование»

| | |
|----------------|-------------------------------|
| Группа: | М8О-208Б-18, №12 |
| Студент: | Коростелев Дмитрий Васильевич |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: | |
| Дата: | 23.12.2019 |

Москва, 2019

1. Задание

Создать 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 (например, координата центра, количество точек и радиус).

Программа должна:

1. Осуществлять ввод из стандартного ввода данных фигур, согласно варианту задания;
2. Программа должна создавать классы, соответствующие введенным данным фигур;
3. Программа должна содержать внутренний буфер, в который помещаются фигуры. Для создания буфера допускается использовать стандартные контейнеры STL. Размер буфера задается параметром командной строки. Например, для буфера размером 10 фигур: `oop_exercise_08 10`
4. При накоплении буфера они должны запускаться на асинхронную обработку, после чего буфер должен очищаться;
5. Обработка должна производиться в отдельном потоке;
6. Реализовать два обработчика, которые должны обрабатывать данные буфера:
 - a. Вывод информации о фигурах в буфере на экран;
 - b. Вывод информации о фигурах в буфере в файл. Для каждого буфера должен создаваться файл с уникальным именем.
7. Оба обработчика должны обрабатывать каждый введенный буфер. Т.е. после каждого заполнения буфера его содержимое должно выводиться как на экран, так и в файл.
8. В программе должно быть ровно два потока (thread). Один основной (main) и второй для обработчиков;
9. В программе должен явно прослеживаться шаблон Publish-Subscribe. Каждый обработчик должен быть реализован как отдельный подписчик.
10. Реализовать в основном потоке (main) ожидание обработки буфера в потоке-обработчике. Т.е. после отправки буфера на обработку основной поток должен ждать, пока поток обработчик выведет данные на экран и запишет в файл.

2. Адрес репозитория на GitHub

https://github.com/Dmitry4K/oop_exercise_08

3. Код программы на C++

Main.cpp

```
#include<iostream>
#include<memory>
#include<vector>
#include<thread>
#include<mutex>
#include<future>
#include<condition_variable>
#include"pentagon.h"
#include"trapeze.h"
#include"rhombus.h"
#include"figure.h"
#include"factory.h"
#include"handler.h"
#include"handlers.h"

void handle(std::vector<std::unique_ptr<figure>>& figures, int buffer_size,
std::condition_variable& readed, std::condition_variable& handled, std::mutex& mtx, bool&
Stop) {
    std::unique_lock<std::mutex> lock(mtx);
    handled.notify_all();
    std::vector<std::unique_ptr<handler>> handlers;

    handlers.push_back(std::make_unique<file_handler>());
    handlers.push_back(std::make_unique<console_handler>());
    while (!(Stop)) {
        readed.wait(lock);
        //std::cout << figures.size() << std::endl;
        for (int i = 0; i < handlers.size(); ++i) {
            handlers[i]->execute(figures);
        }
        figures.clear();
        handled.notify_all();
    }
    return;
}

int main(int argc, char* argv[]) {
    if(argc != 2)
        return 1;
    std::condition_variable readed;
    std::condition_variable handled;
    std::vector<std::unique_ptr<figure>> figures;
    std::unique_ptr<factory> my_factory;
    std::mutex mtx;
    std::unique_lock<std::mutex> lock(mtx);
    int buffer_size, menu;
    buffer_size = std::stoi(argv[1]);
    bool stop = false;
    std::thread handler(handle, std::ref(figures), buffer_size, std::ref(readed),
std::ref(handled),ref(mtx), std::ref(stop));
    handled.wait(lock);
    while (true) {
        for (int i = 0; i < buffer_size; ++i) {
            std::cout << "1. Pentagon" << std::endl;
            std::cout << "2. Rhombus" << std::endl;
            std::cout << "3. Trapeze" << std::endl;
            std::cin >> menu;
            switch (menu) {
                case 1 :
                    my_factory = std::make_unique<pentagon_factory>();
                    figures.push_back(my_factory->build(std::cin));
```

```

        break;
    case 2 :
        my_factory = std::make_unique<rhombus_factory>();
        figures.push_back(my_factory->build(std::cin));
        break;
    case 3 :
        my_factory = std::make_unique<trapeze_factory>();
        figures.push_back(my_factory->build(std::cin));
        break;
    }
}
readed.notify_all();
handled.wait(lock);
std::cout << "Continue? 'y' - Yes 'n' - No" << std::endl;
char answer;
std::cin >> answer;
if (answer != 'y')
    break;
}
stop = true;
readed.notify_all();
lock.unlock();
handler.join();
return 0;
}

```

Pentagon.cpp

```

#include "pentagon.h"
#include <iostream>
#include <fstream>
void pentagon::read(std::istream& is) {
    for (int i = 0; i < 5; i++) {
        is >> vertices[i].x >> vertices[i].y;
    }
}
void pentagon::print(std::ostream& os) const {
    for (int i = 0; i < 5; i++) {
        os << vertices[i].x << ' ' << vertices[i].y << std::endl;
    }
}
void pentagon::print(std::string& filename) const {
    std::ofstream file;
    file.open(filename);
    if (!file.is_open()) {
        std::cerr << "File is not open" << std::endl;
        return;
    }
    file << "pentagon" << std::endl;
    for (int i = 0; i < 5; i++) {
        file << vertices[i].x << ' ' << vertices[i].y << std::endl;
    }
    file.close();
}

```

Rhombus.cpp

```

#include "rhombus.h"
#include <iostream>
#include <fstream>
void rhombus::read(std::istream& is) {
    for (int i = 0; i < 4; i++) {
        is >> vertices[i].x >> vertices[i].y;
    }
}

```

```

void rhombus::print(std::ostream& os) const {
    for (int i = 0; i < 4; i++) {
        os << vertices[i].x << ' ' << vertices[i].y << std::endl;
    }
}
void rhombus::print(std::string& filename) const {
    std::ofstream file;
    file.open(filename);
    if (!file.is_open()) {
        std::cerr << "File is not open" << std::endl;
        return;
    }
    file << "rhombus" << std::endl;
    for (int i = 0; i < 4; i++) {
        file << vertices[i].x << ' ' << vertices[i].y << std::endl;
    }
    file.close();
}

```

Trapeze.cpp

```

#include "trapeze.h"
#include <iostream>
#include <fstream>
void trapeze::read(std::istream& is) {
    for (int i = 0; i < 4; i++) {
        is >> vertices[i].x >> vertices[i].y;
    }
}
void trapeze::print(std::ostream& os) const {
    for (int i = 0; i < 4; i++) {
        os << vertices[i].x << ' ' << vertices[i].y << std::endl;
    }
}
void trapeze::print(std::string& filename) const {
    std::ofstream file;
    file.open(filename);
    if (!file.is_open()) {
        std::cerr << "File is not open" << std::endl;
        return;
    }
    file << "trapeze" << std::endl;
    for (int i = 0; i < 4; i++) {
        file << vertices[i].x << ' ' << vertices[i].y << std::endl;
    }
    file.close();
}

```

Factory.h

```

#pragma once
#include <iostream>
#include "figure.h"
#include "pentagon.h"
#include "rhombus.h"
#include "trapeze.h"
struct factory {
public:
    virtual std::unique_ptr<figure> build(std::istream& is) = 0;
    virtual ~factory() = default;
};

struct pentagon_factory : factory {
    std::unique_ptr<figure> build(std::istream& is) override {
        std::unique_ptr<pentagon> temp;
    }
}

```

```

        temp = std::make_unique<pentagon>();
        temp->read(is);
        return std::move(temp);
    }
};
struct trapeze_factory : factory {
    std::unique_ptr<figure> build(std::istream& is) override {
        std::unique_ptr<trapeze> temp;

        temp = std::make_unique<trapeze>();
        temp->read(is);
        return std::move(temp);
    }
};
struct rhombus_factory : factory {
    std::unique_ptr<figure> build(std::istream& is) override {
        std::unique_ptr<rhombus> temp;

        temp = std::make_unique<rhombus>();
        temp->read(is);
        return std::move(temp);
    }
};

```

Figure.h

```

#pragma once
#include<iostream>
#include<fstream>
struct figure {
    //    figure() = 0;
    virtual void read(std::istream& is) = 0;
    virtual void print(std::ostream& os) const = 0;
    virtual void print(std::string& filename) const = 0;
    virtual ~figure() = default;
};

struct vertex {
    int x, y;
};

```

Handler.h

```

#pragma once
#include<vector>
#include"figure.h"
struct handler {
    virtual void execute(std::vector<std::unique_ptr<figure>>& figures ) = 0;
    virtual ~handler() = default;
};

```

Handlers.h

```

#pragma once
#include"handler.h"
#include<string>
#include<fstream>
#include"figure.h"
struct file_handler : handler {
    void execute(std::vector<std::unique_ptr<figure>>& figures) override {
        static int count_file = 0;
        std::string filename = "";
        ++count_file;
        filename = "file_" + std::to_string(count_file) + ".txt";
        std::ofstream file(filename);
    }
};

```

```

        for (int i = 0; i < figures.size(); ++i) {
            figures[i]->print(file);
        }
    }
};

struct console_handler : handler {
    void execute(std::vector<std::unique_ptr<figure>>& figures) override {
        for (int i = 0; i < figures.size(); ++i) {
            figures[i]->print(std::cout);
        }
    }
};

```

Pentagon.h

```

#pragma once
#include<memory>
#include<array>
#include<string>
#include"figure.h"
struct pentagon : figure {

private:
    std::array<vertex, 5> vertices;//хранилище вершин треугольника
public:
    void read(std::istream& is) override;
    void print(std::ostream& os) const override;
    void print(std::string& filename) const override;
};

```

Rhombus.h

```

#pragma once
#include<memory>
#include<array>
#include<string>
#include"figure.h"
struct rhombus : figure {

private:
    std::array<vertex, 4> vertices;//хранилище вершин треугольника
public:
    void read(std::istream& is) override;
    void print(std::ostream& os) const override;
    void print(std::string& filename) const override;
};

```

Trapeze.h

```

#pragma once
#include<memory>
#include<array>
#include"figure.h"
struct trapeze : figure {
private:
    std::array<vertex, 4> vertices;//хранилище вершин треугольника
public:
    void read(std::istream& is) override;
    void print(std::ostream& os) const override;
    void print(std::string& filename) const override;
};

```

4. Результаты выполнения тестов

Тест 1

1

1. Pentagon

2. Rhombus

3. Trapeze

1

1 1

1 1

1 1

1 1

1 1

1 1

1 1

1 1

1 1

1 1

Continue? 'y' - Yes 'n' - No

y

1. Pentagon

2. Rhombus

3. Trapeze

2

1 1

1 1

1 1

1 1

1 1

1 1

1 1

1 1

Continue? 'y' - Yes 'n' - No

y

1. Pentagon

2. Rhombus

3. Trapeze

3

1 1

1 1

1 1

1 1

1 1

1 1

1 1
1 1
Continue? 'y' - Yes 'n' - No

Тест 2

2
1. Pentagon
2. Rhombus
3. Trapeze

1
1 1
1 1
1 1
1 1
1 1
1. Pentagon
2. Rhombus
3. Trapeze

2
2 1
1 11
1 1
1 1
1 1
1 1
1 1
1 1
1 1
2 1
1 11
1 1
1 1

Continue? 'y' - Yes 'n' - No
н

5. Объяснение результатов работы программы

Программа запрашивает размер буфера, который вводится через стандартный поток ввода, далее, создается дочерний поток с обработчиками, при этом основной поток ждет его создания. Далее заполняется буфер фигурами, после чего буфер отправляется на обработку двумя обработчиками, далее пользователь может выбрать — заполнить буфер еще раз или выйти.

6. Вывод

При разработке программ очень редко задействуется один поток или процесс, так как очень выгодно содержать большое кол-во синхронизированных потоков, на которых происходят вычисления. Для этого в стандартных библиотеках языка C++ присутствуют специальные классы потоков, критических переменных, мьютексов и т.д. для реализации потоков и их синхронизации. Каждый программист должен уметь пользоваться ими и писать многопоточные программы.