Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа №8
по курсу «ООП»

Тема:
Многопоточность.

Студент:	Валов В.В
Группа:	М80-308Б-18
Преподаватель:	Поповкин А.В
Вариант:	
Оценка:	
Дата:	

Москва
2020

1. **Код программы:**

factory.cpp

```
#include "factory.h"
#include "square.h"
#include "rectangle.h"
#include "trapezoid.h"

std::shared_ptr<figure> factory::new_figure(std::istream &is) {
    std::string name;
    is >> name;
    if (name == "square") {
        return std::shared_ptr<figure> ( new square(is));
    } else if ( name == "rectangle") {
        return std::shared_ptr<figure> ( new rectangle(is));
    } else if ( name == "trapezoid") {
        return std::shared_ptr<figure> ( new trapezoid(is));
    } else {
        throw std::logic_error("There's no such figure\n");
    }
}
```

factory.h

```
#ifndef _D_FACTORY_H_
#define _D_FACTORY_H_

#include <memory>
#include <iostream>
#include <fstream>
#include "figure.h"
#include <string>

struct factory {
    std::shared_ptr<figure> new_figure(std::istream& is);
};

#endif // _D_FACTORY_H_
```

figure.h

```
#include <iostream>
#include "point.h"
#include <cmath>
```

```
#ifndef _D_FIGURE_H_
#define _D_FIGURE_H_

struct figure {
   virtual point center() const = 0;
   virtual std::ostream& print(std::ostream& os) const = 0;
   virtual double area() const = 0;
   virtual ~figure() {}
};

#endif //_D_FIGURE_H_
```

main.cpp

```
#include <iostream>
#include <memory>
#include <vector>
#include <thread>

#include "factory.h"
#include "figure.h"
#include "subscriber.h"

void help() {
   std::cout << "help - print this menu\n"
            "add <square, rectangle or trapezoid> <vertices> - add a figure\n"
            "exit\n";
}

int main(int argc,char* argv[]) {
   if (argc != 2) {
      std::cout << "2 arguments needed\n";
      return 1;
   }

   int  buffer_size = std::stoi(argv[1]);
   std::shared_ptr<std::vector<std::shared_ptr<figure>>> buffer =
std::make_shared<std::vector<std::shared_ptr<figure>>>();
   buffer->reserve(buffer_size);
   factory factory;
   std::string cmd;
   subscriber sub;
   sub.processors.push_back(std::make_shared<stream_processor>());
   sub.processors.push_back(std::make_shared<file_processor>());
   std::thread sub_thread(std::ref(sub));
```

```
    while (true) {
        std::unique_lock<std::mutex> locker(sub.mtx);
        std::cin >> cmd;
        if (cmd == "help") {
            help();
        } else if (cmd == "add") {
            try {
                buffer->push_back(factory.new_figure(std::cin));
            } catch (std::logic_error &e) {
                std::cout << e.what() << '\n';
                continue;
            }
            if (buffer->size() == buffer_size) {
                std::cout << "You've reached the limit\n";
                sub.buffer = buffer;
                sub.cond_var.notify_all();
                sub.cond_var.wait(locker, [&](){ return sub.buffer == nullptr;});
                buffer->clear();
            }
        } else if (cmd == "quit")  {
            break;
        } else {
            std::cout << "Wrong command\n";
        }
    }
    sub.stop = true;
    sub.cond_var.notify_all();
    sub_thread.join();
    return 0;
}
```

point.cpp

```
#include "point.h"

point operator+ (point lhs, point rhs) {
    return {lhs.x + rhs.x, lhs.y + rhs.y};
}

point operator- (point lhs, point rhs) {
    return {lhs.x - rhs.x, lhs.y - rhs.y};
}

point operator/ (point p, double t) {
    return {p.x / t, p.y / t};
}

std::istream &operator>>(std::istream &is, point &p) {
    is >> p.x >> p.y;
    return is;
}

std::ostream &operator<< (std::ostream &os, const point &p ) {
    os << p.x << " " << p.y << std::endl;
    return os;
}
```

processor.cpp

```
#include "processor.h"

void
stream_processor::process(std::shared_ptr<std::vector<std::shared_ptr<figure>>>
buffer) {
    for (auto figure : *buffer) {
        figure->print(std::cout);
    }
}

void
file_processor::process(std::shared_ptr<std::vector<std::shared_ptr<figure>>>
buffer) {
    std::ofstream fout;
    fout.open(std::to_string(cnt) + ".txt");
    cnt++;
    if (!fout.is_open()) {
        std::cout << "can't open\n";
        return;
    }
    for (auto figure : *buffer) {
        figure->print(fout);
    }

}
```

processor.h

```
#ifndef _D_PROCESSOR_H_
#define _D_PROCESSOR_H_

#include <iostream>
```

```
#include <condition_variable>
#include <thread>
#include <vector>
#include <mutex>

#include "factory.h"
#include "figure.h"

struct processor {
   virtual void process(std::shared_ptr<std::vector<std::shared_ptr<figure>>>
buffer) = 0;
};

struct stream_processor : processor {
   void process(std::shared_ptr<std::vector<std::shared_ptr<figure>>> buffer)
override;
};

struct file_processor : processor {
   void process(std::shared_ptr<std::vector<std::shared_ptr<figure>>> buffer)
override;
private:
   int cnt = 0;
};

#endif // _D_PROCESSOR_H_
```

rectangle.cpp

```
#include "rectangle.h"

rectangle::rectangle(std::istream& is) {
   is >> a1 >> a2 >> a3 >> a4;
}

double rectangle::area() const {
   double xHeight = a2.x - a1.x;
   double yHeight = a2.y - a1.y;

   double xWidth = a3.x - a2.x;
   double yWidth = a3.y - a2.y;

   return sqrt(xHeight * xHeight + yHeight * yHeight) * sqrt(xWidth * xWidth +
yWidth * yWidth);
}
```

```
point rectangle::center() const {
   double x,y;
   x = (a1.x + a2.x + a3.x + a4.x) / 4;
   y = (a1.y + a2.y + a3.y + a4.y) / 4;
   point p(x,y);
   return p;
}

std::ostream& rectangle::print(std::ostream& os) const {
   os << "Rectangle\n"<< a1 << a2 << a3 << a4;
   os << "Center: " << center() << "Area:" << area() << '\n';
   return os;
}
```

rectangle.h

```
#ifndef _D_RECTANGLE_H_
#define _D_RECTANGLE_H_

#include "figure.h"

class rectangle : public figure {
public:
   rectangle() = default;
   rectangle(std::istream& is) ;
   double area() const override;
   point center() const override;
   std::ostream& print(std::ostream& os) const override;
private:
   point a1, a2, a3, a4;
};

#endif // _D_RECTANGLE_H_
```

square.cpp

```
#include "square.h"

square::square(std::istream& is) {
   is >> a1 >> a2 >> a3 >> a4;
}

double square::area() const {
   double vecX = a2.x - a1.x;
   double vecY = a2.y - a1.y;
   return vecX * vecX + vecY * vecY;
}

point square::center() const {
   double x,y;
   x = (a1.x + a2.x + a3.x + a4.x) / 4;
   y = (a1.y + a2.y + a3.y + a4.y) / 4;
   point p(x,y);
   return p;
}

std::ostream& square::print(std::ostream& os) const {
   os << "Square\n"<< a1 << a2 << a3 << a4;
   os << "Center: " << center() << "Area:" << area() << '\n';
   return os;
}
```

square.h

```
#ifndef _D_SQUARE_H_
#define _D_SQUARE_H_

#include "figure.h"

class square : public figure {
public:
    square() = default;
    square(std::istream& is);
    double area() const override;
    point center() const override;
    std::ostream& print(std::ostream&) const override;
private:
    point a1, a2, a3, a4;
};

#endif // _D_SQUARE_H_
```

subscriber.cpp

```
#include "subscriber.h"

void subscriber::operator()() {
    for(;;) {
        std::unique_lock<std::mutex>lock(mtx);
        cond_var.wait(lock,[&]{ return (buffer != nullptr || stop);});
        if (stop) {
            break;
        }
        for (auto elem: processors) {
            elem->process(buffer);
        }
        buffer = nullptr;
        cond_var.notify_all();
    }
}
```

subscriber.h

```
#ifndef _D_SUBSCTIBER_H_
#define _D_SUBSCTIBER_H_

#include <iostream>
#include <condition_variable>
#include <thread>
#include <vector>
#include <mutex>
#include "factory.h"
#include "figure.h"
#include "processor.h"

struct subscriber {
    void operator()();
    std::vector<std::shared_ptr<processor>> processors;
    std::shared_ptr<std::vector<std::shared_ptr<figure>>> buffer;
    std::mutex mtx;
    std::condition_variable cond_var;
    bool stop = false;
};

#endif // _D_SUBSCTIBER_H_
```

meson.build

```
project('oop_exercise_08', 'cpp')

add_project_arguments('-std=c++17', '-w', '-pthread', language : 'cpp')

thread_dep = dependency('threads')

executable(
    meson.project_name(),
    'main.cpp',
    'factory.cpp',
    'point.cpp',
    'processor.cpp',
    'rectangle.cpp',
    'square.cpp',
    'subscriber.cpp',
    'trapezoid.cpp',
    dependencies : thread_dep
)
```

2. **Набор testcases:**

test_01.test

```
add square 0 0 0 0 0 0 0 0
add rectangle 1 1 1 1 1 1 1 1
add trapezoid 2 2 2 2 2 2 2 2
quit
```

test_02.test

```
add square 2 2 2 2 2 2 2 2
add rectangle  3 3 3 3 3 3 3 3
add square  4 4 4 4 4 4 4 4
add trapezoid 5 5 5 5 5 5 5 5
add square 2 2 2 2 2 2 2 2
add rectangle 3 3 3 3 3 3 3 3
add square 5 5 5 5 5 5 5 5
add rectangle 6 6 6 6 6 6 6 6
quit
```

test_01.result

```
You've reached the limit
Square
0 0
0 0
0 0
0 0
Center: 0 0
Area:0
Rectangle
1 1
1 1
```

```
1 1
1 1
Center: 1 1
Area:0
Trapezoid
2 2
2 2
2 2
2 2
Center: 2 2
Area:0
```

test_02.result

```
You've reached the limit
Square
2 2
2 2
2 2
2 2
Center: 2 2
Area:0
Rectangle
3 3
3 3
3 3
3 3
Center: 3 3
Area:0
Square
4 4
4 4
4 4
4 4
Center: 4 4
Area:0
You've reached the limit
Trapezoid
5 5
5 5
5 5
5 5
Center: 5 5
Area:0
Square
2 2
```

```
2 2
2 2
2 2
Center: 2 2
Area:0
Rectangle
3 3
3 3
3 3
3 3
Center: 3 3
Area:0
```

3. Объяснение результатов работы программы:

При запуске программы пользователь задаёт размер буфера, в который помещаются задаваемые им фигуры. Когда буфер становится полным, в терминал выводится вся информация о фигурах, а буфер очищается.

4. Вывод:

Выполняя данную лабораторную, я получил опыт работы с потоками в C++.