Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Отчёт по Лабораторной работе №7 “Проектирование, структуры классов“ по курсу “Объектно-Ориентированное Программирование“ III Семестр

Студент:	Катермин В.С.
Группа:	М8О-208Б-18
Преподаватель:	Журавлёв А.А.
Оценка:	
Дата:	**09.12.19**

1. **Тема**: Проектирование, Структуры классов в C++.
2. **Код программы:**

vertex.h

```
#ifndef D_VERTEX_H
#define D_VERTEX_H

struct vertex {
   int32_t x, y;

};

#endif //D_VERTEX_H
```

figure.h

```
#ifndef D_FIGURE_H
#define D_FIGURE_H

#include <iostream>
#include <memory>
#include <array>
#include <cmath>

#include "sdl.h"
#include "imgui.h"
#include "vertex.h"

struct color {
   color() : r(255), g(255), b(255) {}
   int32_t r, g, b;
   color(int r_, int g_, int b_) :r(r_), g(g_), b(b_) {}
   void set_color(int r_, int g_, int b_) { r = r_, g = g_, b = b_; }
};

struct figure {
   virtual void render(const sdl::renderer& renderer) const = 0;
   virtual void save(std::ostream& os) const = 0;
   virtual bool erase_check(const vertex& v) const = 0;
   virtual ~figure() = default;

   color color_{};
   virtual void set_color(int r, int g, int b) {
      color_.r = r;
      color_.g = g;
      color_.b = b;
   }
};


#endif //D_FIGURE_H
```

triangle.h

```
#ifndef D_TRIANGLE_H
#define D_TRIANGLE_H

#include "figure.h"

struct triangle : figure {
   triangle(const std::array<vertex, 3>& vertices) : vertices_(vertices) {}

   void render(const sdl::renderer& renderer) const override {
```

```
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 3; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
                vertices_[(i + 1) % 3].x, vertices_[(i + 1) % 3].y);
        }
    }

    void save(std::ostream& os) const override {
        os << "triangle\n";
        for (int32_t i = 0; i < 3; ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }
        os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
    }

    bool  erase_check(const vertex& v) const override {
        int32_t j;
        bool count = false;
        for (int32_t i = 0; i < 3; ++i) {
            for (i = 0, j = vertices_.size() - 1; i < vertices_.size(); j = i++) {
                if (((vertices_[i].y > v.y) != (vertices_[j].y > v.y)) && (v.x < (vertices_[j].x - vertices_[i].x) * (v.y -
vertices_[i].y) / (vertices_[j].y - vertices_[i].y) + vertices_[i].x)) {
                    count = !count;
                }
            }
        }
        return count;
    }

private:
    std::array<vertex, 3> vertices_;

};
#endif //D_TRIANGLE_H
```

square.h

```
#ifndef D_SQUARE_H
#define D_SQUARE_H

#include "figure.h"

struct square : figure {
    square(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 4; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
                vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
        }
    }

    void save(std::ostream& os) const override {
        os << "square\n";
        for (int32_t i = 0; i < 4; ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }
        os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
    }

    bool  erase_check(const vertex& v) const override {
        int32_t j;
        bool count = false;
```

```
        for (int32_t i = 0; i < 4; ++i) {
            for (i = 0, j = vertices_.size() - 1; i < vertices_.size(); j = i++) {
                if (((vertices_[i].y > v.y) != (vertices_[j].y > v.y)) && (v.x < (vertices_[j].x - vertices_[i].x) * (v.y -
vertices_[i].y) / (vertices_[j].y - vertices_[i].y) + vertices_[i].x)) {
                    count = !count;
                }
            }
        }
        return count;
    }

private:
    std::array<vertex, 4> vertices_;

};
#endif //D_SQUARE_H
```

rectangle.h

```
#ifndef D_RECTANGLE_H
#define D_RECTANGLE_H

#include "figure.h"

struct rectangle : figure {
    rectangle(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 4; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
                vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
        }
    }

    void save(std::ostream& os) const override {
        os << "rectangle\n";
        for (int32_t i = 0; i < 4; ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }
        os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
    }

    bool  erase_check(const vertex& v) const override {
        int32_t j;
        bool count = false;
        for (int32_t i = 0; i < 4; ++i) {
            for (i = 0, j = vertices_.size() - 1; i < vertices_.size(); j = i++) {
                if (((vertices_[i].y > v.y) != (vertices_[j].y > v.y)) && (v.x < (vertices_[j].x - vertices_[i].x) * (v.y -
vertices_[i].y) / (vertices_[j].y - vertices_[i].y) + vertices_[i].x)) {
                    count = !count;
                }
            }
        }
        return count;
    }

private:
    std::array<vertex, 4> vertices_;

};
#endif //D_RECTANGLE_H
```

trapezoid.h

```
#ifndef D_TRAPEZOID_H
#define D_TRAPEZOID_H

#include "figure.h"

struct trapezoid : figure {
   trapezoid(const std::array<vertex, 4>& vertices) : vertices_(vertices) {}

   void render(const sdl::renderer& renderer) const override {
      renderer.set_color(color_.r, color_.g, color_.b);
      for (int32_t i = 0; i < 4; ++i) {
         renderer.draw_line(vertices_[i].x, vertices_[i].y,
            vertices_[(i + 1) % 4].x, vertices_[(i + 1) % 4].y);
      }
   }

   void save(std::ostream& os) const override {
      os << "trapezoid\n";
      for (int32_t i = 0; i < 4; ++i) {
         os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
      }
      os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
   }

   bool  erase_check(const vertex& v) const override {
      int32_t j;
      bool count = false;
      for (int32_t i = 0; i < 4; ++i) {
         for (i = 0, j = vertices_.size() - 1; i < vertices_.size(); j = i++) {
            if (((vertices_[i].y > v.y) != (vertices_[j].y > v.y)) && (v.x < (vertices_[j].x - vertices_[i].x) * (v.y -
vertices_[i].y) / (vertices_[j].y - vertices_[i].y) + vertices_[i].x)) {
               count = !count;
            }
         }
      }
      return count;
   }

private:
   std::array<vertex, 4> vertices_;

};
#endif //D_TRAPEZOID_H
```

polyline.h

```
#ifndef D_POLYLINE_H
#define D_POLYLINE_H

#include "figure.h"

struct  polyline : figure {
   polyline(const std::vector<vertex>& vertices) : vertices_(vertices) {}

   void render(const sdl::renderer& renderer) const override {
      renderer.set_color(color_.r, color_.g, color_.b);
      for (int32_t i = 0; i < vertices_.size() - 1; ++i) {
         renderer.draw_line(vertices_[i].x, vertices_[i].y,
            vertices_[(i + 1)].x, vertices_[(i + 1)].y);
      }
   }
```

```
    void save(std::ostream& os) const override {
        os << "polyline" << ' ' << vertices_.size() << std::endl;
        for (int32_t i = 0; i < vertices_.size(); ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }
        os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
    }

    bool  erase_check(const vertex& v) const override {
        int32_t j;
        bool count = false;
        for (int32_t i = 0; i < vertices_.size(); ++i) {
            for (i = 0, j = vertices_.size() - 1; i < vertices_.size(); j = i++) {
                if (((vertices_[i].y > v.y) != (vertices_[j].y > v.y)) && (v.x == (vertices_[j].x - vertices_[i].x) * (v.y -
vertices_[i].y) / (vertices_[j].y - vertices_[i].y) + vertices_[i].x)) {
                    count = !count;
                }
            }
        }
        return count;
    }

private:
    std::vector<vertex> vertices_;

};
#endif //D_POLYLINE_H
```

polygon.h

```
#ifndef D_POLYGON_H
#define D_POLYGON_H

#include "figure.h"

struct  polygon : figure {
    polygon(const std::vector<vertex>& vertices) : vertices_(vertices) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < vertices_.size() - 1; ++i) {
            renderer.draw_line(vertices_[i].x, vertices_[i].y,
                vertices_[(i + 1)].x, vertices_[(i + 1)].y);
        }
    }

    void save(std::ostream& os) const override {
        os << "polygon" << ' ' << vertices_.size() << std::endl;
        for (int32_t i = 0; i < vertices_.size(); ++i) {
            os << vertices_[i].x << ' ' << vertices_[i].y << '\n';
        }
        os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
    }

    bool  erase_check(const vertex& v) const override {
        int32_t j;
        bool count = false;
        for (int32_t i = 0; i < vertices_.size(); ++i) {
            for (i = 0, j = vertices_.size() - 1; i < vertices_.size(); j = i++) {
                if (((vertices_[i].y > v.y) != (vertices_[j].y > v.y)) && (v.x < (vertices_[j].x - vertices_[i].x) * (v.y -
vertices_[i].y) / (vertices_[j].y - vertices_[i].y) + vertices_[i].x)) {
                    count = !count;
                }
            }
```

```
        }
        return count;
    }

private:
    std::vector<vertex> vertices_;

};
#endif //D_POLYGON_H
```

circle.h

```
#ifndef D_CIRCLE_H
#define D_CIRCLE_H

#include "figure.h"

struct  circle : figure {
    circle(const vertex& center, const double& radius) : center_(center), radius_(radius) {}

    void render(const sdl::renderer& renderer) const override {
        renderer.set_color(color_.r, color_.g, color_.b);
        for (int32_t i = 0; i < 360; ++i) {
            double rx1 = center_.x + radius_ * cos(i * (M_PI / 180));
            double ry1 = center_.y + radius_ * sin(i * (M_PI / 180));
            double rx2 = center_.x + radius_ * cos((i + 1) * (M_PI / 180));
            double ry2 = center_.y + radius_ * sin((i + 1) * (M_PI / 180));
            //renderer.draw_line(vertices_[0].x, vertices_[0].y, rx1, ry1);
            renderer.draw_line(rx1, ry1, rx2, ry2);
        }
    }

    void save(std::ostream& os) const override {
        os << "circle" << std::endl;
        os << center_.x << ' ' << center_.y << ' ' << radius_ << '\n';

        os << this->color_.r << ' ' << this->color_.g << ' ' << this->color_.b << std::endl;
    }

    bool erase_check(const vertex& v) const override {
        int32_t lx = v.x - center_.x;
        int32_t ly = v.y - center_.y;
        double l = sqrt(lx * lx + ly * ly);
        if (l <= radius_) {
            return true;
        }
        else {
            return false;
        }
    }

private:
    vertex center_;
    double radius_;
};

#endif //D_CIRCLE_H
```

document.h

```
#ifndef D_DOCUMENT_H_
#define D_DOCUMENT_H_

#include<string>
```

```
#include<vector>
#include<memory>
#include<stack>

#include "figure.h"
#include "triangle.h"
#include "square.h"
#include "rectangle.h"
#include "trapezoid.h"
#include "polyline.h"
#include "polygon.h"
#include "circle.h"

struct command {
	virtual void undo() = 0;
	virtual ~command() = default; //Деструктор (пока под вопросом)
};

struct document {
	document() {};
	void add_fgrs(std::unique_ptr<figure> fgr);
	void rmv_fgrs(int32_t rmv_id);
	void undo();

	std::vector<std::unique_ptr<figure>> figures;
	std::stack<std::unique_ptr<command>> commands;
};

#endif // D_DOCUMENT_H_
```

document.cpp

```
#include "document.h"

struct add_cmd : command {
	add_cmd(document* document) : document_(document) {}

	void undo() override {
		document_->figures.pop_back();
	}

private:
	document* document_;
	//int32_t idx_;
	//std::unique_ptr<figure> figure_;
};

struct rmv_cmd : command {
	rmv_cmd(document* document, int32_t idx, std::unique_ptr<figure>&& figure) : document_(document),
idx_(idx), figure_(std::move(figure)) {}

	void undo() override {
		//document_->figures[idx_] = std::move(figure_);
		document_->figures.emplace(document_->figures.begin() + idx_, std::move(figure_));
		//document_->figures.pop_back();
	}

private:
	document* document_;
	std::unique_ptr<figure> figure_;
	int32_t idx_;
};

void document::add_fgrs(std::unique_ptr<figure> fgr) {
```

```
    figures.emplace_back(std::move(fgr)); //добавить полученный результат в вектор фигур
    commands.push(std::make_unique<add_cmd>(this));
}

void document::rmv_fgrs(int32_t rmv_id) {
    commands.push(std::make_unique<rmv_cmd>(this, rmv_id, std::move(figures[rmv_id])));
    figures.erase(figures.begin() + rmv_id);
}

void document::undo() {
    if (commands.size()) {
        commands.top()->undo();
        commands.pop();
    }
}
```

painter.h

```
#ifndef D_PAINTER_H
#define D_PAINTER_H

#include <array>
#include <fstream>
#include <memory>
#include <vector>
#include <cmath>

#include "sdl.h"
#include "imgui.h"
#include "figure.h"
#include "triangle.h"
#include "square.h"
#include "rectangle.h"
#include "trapezoid.h"
#include "circle.h"
#include "polyline.h"
#include "polygon.h"
#include "document.h"

struct builder {
   virtual std::unique_ptr<figure> add_vertex(const vertex& v) = 0; // добавление новой вершины в фигуру

   virtual ~builder() = default; // деструктор (Не нужен, но должен быть)

};

struct poly_builder {
   virtual std::unique_ptr<figure> add_vertex(const vertex& v) = 0; // добавление новой вершины в поли-фигуру
   virtual std::unique_ptr<figure> finish_it(const vertex& v) = 0; // завершение построение поли-фигуры

   virtual ~poly_builder() = default; // деструктор (Не нужен, но должен быть)

};

struct triangle_builder : builder {
   std::unique_ptr<figure> add_vertex(const vertex& v) {
      vertices_[n_] = v;
      n_ += 1;
      if (n_ != 3) {
         return nullptr;
      }
      return std::make_unique<triangle>(vertices_);
   }
```

```
private:
   int32_t n_ = 0;
   std::array<vertex, 3> vertices_; // вершины фигуры

};

struct square_builder : builder {
   std::unique_ptr<figure> add_vertex(const vertex& v) {
      if (n_ == 2) {
         int32_t vx = vertices_[1].x - vertices_[0].x;
         int32_t vy = vertices_[1].y - vertices_[0].y;
         int32_t D = (v.x - vertices_[0].x) * vy - (v.y - vertices_[0].y) * vx;
         if (D < 0) {
            vertices_[n_] = vertex{ vertices_[1].x - vy, vertices_[1].y + vx };
            n_ += 1;
            vertices_[n_] = vertex{ vertices_[0].x - vy, vertices_[0].y + vx };
            n_ += 1;
         }
         else {
            vertices_[n_] = vertex{ vertices_[1].x + vy, vertices_[1].y - vx };
            n_ += 1;
            vertices_[n_] = vertex{ vertices_[0].x + vy, vertices_[0].y - vx };
            n_ += 1;
         }
      }
      else {
         vertices_[n_] = v;
         n_ += 1;
      }
      if (n_ != 4) {
         return nullptr;
      }
      return std::make_unique<square>(vertices_);
   }

private:
   int32_t n_ = 0;
   std::array<vertex, 4> vertices_; // вершины фигуры

};

struct rectangle_builder : builder {
   std::unique_ptr<figure> add_vertex(const vertex& v) {
      if (n_ == 2) {
         int32_t vx1 = vertices_[1].x - vertices_[0].x;
         int32_t vy1 = vertices_[1].y - vertices_[0].y;
         int32_t px = ((vx1 * vy1 * (v.y - vertices_[0].y) + vertices_[0].x * pow(vy1, 2) + v.x * pow(vx1, 2)) /
(pow(vy1, 2) + pow(vx1, 2)));
         int32_t py = (vy1 * (px - vertices_[0].x)) / (vx1)+vertices_[0].y;
         int32_t vx2 = v.x - px;
         int32_t vy2 = v.y - py;

         vertices_[n_] = vertex{ vertices_[1].x + vx2, vertices_[1].y + vy2 };
         n_ += 1;
         vertices_[n_] = vertex{ vertices_[0].x + vx2, vertices_[0].y + vy2 };
         n_ += 1;
      }
      else {
         vertices_[n_] = v;
         n_ += 1;
      }
      if (n_ != 4) {
         return nullptr;
      }
```

```
        return std::make_unique<rectangle>(vertices_);
    }

private:
    int32_t n_ = 0;
    std::array<vertex, 4> vertices_; // вершины фигуры

};

struct trapezoid_builder : builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        if (n_ == 2) {
            int32_t vx1 = vertices_[1].x - vertices_[0].x;
            int32_t vy1 = vertices_[1].y - vertices_[0].y;
            int32_t px = ((vx1 * vy1 * (v.y - vertices_[0].y) + vertices_[0].x * pow(vy1, 2) + v.x * pow(vx1, 2)) /
(pow(vy1, 2) + pow(vx1, 2)));
            int32_t py = (vy1 * (px - vertices_[0].x)) / (vx1)+vertices_[0].y;
            int32_t vx2 = v.x - px;
            int32_t vy2 = v.y - py;
            int32_t vx3 = vertices_[1].x - px;
            int32_t vy3 = vertices_[1].y - py;
            int32_t fx = vertices_[0].x + vx2 + vx3;
            int32_t fy = vertices_[0].y + vy2 + vy3;

            vertices_[n_] = vertex{ v.x, v.y };
            n_ += 1;
            vertices_[n_] = vertex{ fx, fy };
            n_ += 1;
        }
        else {
            vertices_[n_] = v;
            n_ += 1;
        }
        if (n_ != 4) {
            return nullptr;
        }
        return std::make_unique<trapezoid>(vertices_);
    }

private:
    int32_t n_ = 0;
    std::array<vertex, 4> vertices_; // вершины фигуры

};

struct  circle_builder : builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        if (n_ == 0) {
            center_ = v;
            n_ += 1;
        }
        else if (n_ == 1) {
            int32_t rx = v.x - center_.x;
            int32_t ry = v.y - center_.y;
            radius_ = sqrt(rx * rx + ry * ry);
            n_ += 1;
        }
        if (n_ != 2) {
            return nullptr;
        }
        return std::make_unique<circle>(center_, radius_);
    }
private:
    int32_t n_ = 0;
```

```
    vertex center_; // центр круга
    double radius_; // радиус круга
};

struct  polyline_builder : poly_builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        vertices_.push_back(v);
        n_ += 1;
        return nullptr;
    }

    std::unique_ptr<figure> finish_it(const vertex& v) {
        vertices_.push_back(v);
        if (n_ < 2) {
            return nullptr;
        }
        return std::make_unique<polyline>(vertices_);
    }

private:
    int32_t n_ = 0;
    std::vector<vertex> vertices_; // вершины фигуры
};

struct  polygon_builder : poly_builder {
    std::unique_ptr<figure> add_vertex(const vertex& v) {
        vertices_.push_back(v);
        n_ += 1;
        return nullptr;
    }

    std::unique_ptr<figure> finish_it(const vertex& v) {
        vertices_.push_back(vertex{ vertices_[0].x, vertices_[0].y });
        if (n_ < 2) {
            return nullptr;
        }
        return std::make_unique<polygon>(vertices_);
    }

private:
    int32_t n_ = 0;
    std::vector<vertex> vertices_; // вершины фигуры
};

#endif //D_PAINTER_H
```

loader.h

```
#ifndef D_LOADER_H
#define D_LOADER_H

#include<vector>
#include<memory>

#include "figure.h"
#include "document.h"

struct loader {
    std::vector<std::unique_ptr<figure>> load(std::ifstream& is) {
        std::string figure_name;
        std::vector<std::unique_ptr<figure>> figures;
        while (is >> figure_name) {
            vertex v;
            if (figure_name == std::string("triangle")) {
```

```
    std::array<vertex, 3> vertices;
    for (int32_t i = 0; i < 3; ++i) {
        is >> v.x >> v.y;
        vertices[i] = v;
    }
    struct color load_clr {};
    is >> load_clr.r >> load_clr.g >> load_clr.b;
    figures.emplace_back(std::make_unique<triangle>(vertices));
    (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);

}
if (figure_name == std::string("square")) {
    std::array<vertex, 4> vertices;
    for (int32_t i = 0; i < 4; ++i) {
        is >> v.x >> v.y;
        vertices[i] = v;
    }
    struct color load_clr {};
    is >> load_clr.r >> load_clr.g >> load_clr.b;
    figures.emplace_back(std::make_unique<square>(vertices));
    (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);
}
else if (figure_name == std::string("rectangle")) {
    std::array<vertex, 4> vertices;
    for (int32_t i = 0; i < 4; ++i) {
        is >> v.x >> v.y;
        vertices[i] = v;
    }
    struct color load_clr {};
    is >> load_clr.r >> load_clr.g >> load_clr.b;
    figures.emplace_back(std::make_unique<rectangle>(vertices));
    (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);
}
else if (figure_name == std::string("trapezoid")) {
    std::array<vertex, 4> vertices;
    for (int32_t i = 0; i < 4; ++i) {
        is >> v.x >> v.y;
        vertices[i] = v;
    }
    struct color load_clr {};
    is >> load_clr.r >> load_clr.g >> load_clr.b;
    figures.emplace_back(std::make_unique<trapezoid>(vertices));
    (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);
}
else if (figure_name == std::string("circle")) {
    vertex center;
    double radius;
    is >> v.x >> v.y >> radius;
        center = v;
    struct color load_clr {};
    is >> load_clr.r >> load_clr.g >> load_clr.b;
    figures.emplace_back(std::make_unique<circle>(center, radius));
    (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);
}
else if (figure_name == std::string("polyline")) {
    std::vector<vertex> vertices;
    int count_v;
    is >> count_v;
    for (int i = 0; i < count_v; ++i) {
        is >> v.x >> v.y;
        vertices.push_back(v);
    }
    struct color load_clr {};
    is >> load_clr.r >> load_clr.g >> load_clr.b;
```

```
            figures.emplace_back(std::make_unique<polyline>(vertices));
            (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);
        }
        else if (figure_name == std::string("polygon")) {
            std::vector<vertex> vertices;
            int count_v;
            is >> count_v;
            for (int i = 0; i < count_v; ++i) {
                is >> v.x >> v.y;
                vertices.push_back(v);
            }
            struct color load_clr {};
            is >> load_clr.r >> load_clr.g >> load_clr.b;
            figures.emplace_back(std::make_unique<polygon>(vertices));
            (*figures[figures.size() - 1]).set_color(load_clr.r, load_clr.g, load_clr.b);
        }
    }
    return figures;
  }
  ~loader() = default; // Деструктор (Не нужен, но должен быть)
};

#endif //D_LOADER_H
```

main.cpp

```
#include <array>
#include <fstream>
#include <iostream>
#include <memory>
#include <vector>

#include "sdl.h"
#include "imgui.h"
#include "figure.h"
#include "triangle.h"
#include "painter.h"
#include "loader.h"
#include "document.h"

int main() {
  document document;
  color fgr_clr{};
  //std::unique_ptr<document> document;
  sdl::renderer renderer("Editor");
  bool quit = false;
  std::unique_ptr<builder> active_builder = nullptr;
  std::unique_ptr<poly_builder> active_poly_builder = nullptr;
  const int32_t file_name_length = 128;
  char file_name[file_name_length] = "";
  int32_t remove_id = 0;
  while (!quit) {
    renderer.set_color(0, 0, 0);
    renderer.clear();

    sdl::event event;

    while (sdl::event::poll(event)) {
      sdl::quit_event quit_event;
      sdl::mouse_button_event mouse_button_event;
      if (event.extract(quit_event)) {
        quit = true;
        break;
```

```
        }
        else if (event.extract(mouse_button_event)) {
            if (active_builder && mouse_button_event.button() == sdl::mouse_button_event::left &&
                mouse_button_event.type() == sdl::mouse_button_event::down) { //Если есть строитель и ЛКМ
                std::unique_ptr<figure> figure = //если в строителе достаточное количество вершин, будет фигура,
иначе nullptr
                    active_builder->add_vertex(vertex{ mouse_button_event.x(), mouse_button_event.y() }); //
добавляем вершины
                if (figure) {
                    //figures.emplace_back(std::move(figure)); //добавить полученный результат в вектор фигур
                    (*figure).set_color(fgr_clr.r, fgr_clr.g, fgr_clr.b);
                    document.add_fgrs(std::move(figure));
                    active_builder = nullptr;
                }
            }
            else if (active_poly_builder) {
                std::unique_ptr<figure> p_figure;
                if (mouse_button_event.button() == sdl::mouse_button_event::left &&
                    mouse_button_event.type() == sdl::mouse_button_event::down) { //Если есть строитель и ЛКМ
                    p_figure = active_poly_builder->add_vertex(vertex{ mouse_button_event.x(),
mouse_button_event.y() });  // В этом случае nullptr
                }
                else if (mouse_button_event.button() == sdl::mouse_button_event::right &&
                    mouse_button_event.type() == sdl::mouse_button_event::down) { //Если есть строитель и ПКМ
                    p_figure = active_poly_builder->finish_it(vertex{ mouse_button_event.x(),
mouse_button_event.y() }); // завершение поли-фигуры
                    if (p_figure) {
                        (*p_figure).set_color(fgr_clr.r, fgr_clr.g, fgr_clr.b);
                        document.add_fgrs(std::move(p_figure));
                        active_poly_builder = nullptr;
                    }
                }
            }
            else {
                if (mouse_button_event.button() == sdl::mouse_button_event::left &&
                    mouse_button_event.type() == sdl::mouse_button_event::down) {
                    for (int i = 0; i < document.figures.size(); ++i) {
                        if (document.figures[i]) {
                            if (document.figures[i]->erase_check(vertex{ mouse_button_event.x(),
mouse_button_event.y() })) {
                                if (i < document.figures.size()) {
                                    if (document.figures[i]) {
                                        document.rmv_fgrs(i);
                                    }
                                    //rmv_fgrs(figures.erase(figures.begin() + remove_id);
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    //for (const std::unique_ptr<figure>& figure : document->figures) {
        //figure->render(renderer);
    //}

    for (int i = 0; i < document.figures.size(); ++i) {
        if (document.figures[i]) {
            document.figures[i]->render(renderer);
        }
    }
```

```
ImGui::Begin("Menu");
if (ImGui::Button("New canvas")) {
    document.figures.clear();
}
ImGui::InputText("File name", file_name, file_name_length - 1);
if (ImGui::Button("Save")) {
    std::ofstream os(file_name);
    if (os) {
        for (const std::unique_ptr<figure>& figure : document.figures) {
            figure->save(os);
        }
    }
}
ImGui::SameLine();
if (ImGui::Button("Load")) {
    std::ifstream is(file_name);
    if (is) {
        loader loader;
        document.figures = loader.load(is);
    }
}
ImGui::InputInt("R", &fgr_clr.r);
ImGui::InputInt("G", &fgr_clr.g);
ImGui::InputInt("B", &fgr_clr.b);
if (ImGui::Button("Red")) {
    fgr_clr.set_color(255, 0, 0);
}
ImGui::SameLine();
if (ImGui::Button("Green")) {
    fgr_clr.set_color(0, 255, 0);
}
ImGui::SameLine();
if (ImGui::Button("Blue")) {
    fgr_clr.set_color(0, 0, 255);
}
if (ImGui::Button("Yellow")) {
    fgr_clr.set_color(255, 255, 0);
}
ImGui::SameLine();
if (ImGui::Button("Cyan")) {
    fgr_clr.set_color(0, 255, 255);
}
ImGui::SameLine();
if (ImGui::Button("Magenta")) {
    fgr_clr.set_color(255, 0, 255);
}
if (ImGui::Button("White")) {
    fgr_clr.set_color(255, 255, 255);
}
if (ImGui::Button("Triangle")) {
    active_builder = std::make_unique<triangle_builder>();
}
if (ImGui::Button("Square")) {
    active_builder = std::make_unique<square_builder>();
}
if (ImGui::Button("Rectangle")) {
    active_builder = std::make_unique<rectangle_builder>();
}
if (ImGui::Button("Trapezoid")) {
    active_builder = std::make_unique<trapezoid_builder>();
}
if (ImGui::Button("Circle")) {
    active_builder = std::make_unique<circle_builder>();
}
```

```
        if (ImGui::Button("Poly-Line")) {
            active_poly_builder = std::make_unique<polyline_builder>();
        }
        if (ImGui::Button("Polygon")) {
            active_poly_builder = std::make_unique<polygon_builder>();
        }
        ImGui::InputInt("Remove id", &remove_id);
        if (ImGui::Button("Remove")) {
            if (remove_id < document.figures.size()) {
                if (document.figures[remove_id]) {
                    document.rmv_fgrs(remove_id);
                }
                //rmv_fgrs(figures.erase(figures.begin() + remove_id);
            }
        }
        if (ImGui::Button("Undo")) {
            document.undo();
        }

        ImGui::End();

        renderer.present();
    }
}
```

Makefile

```
cmake_minimum_required(VERSION 3.0)

project(lab7)

set(CMAKE_CXX_STANDARD_REQUIRED YES)
set(CMAKE_CXX_STANDARD 17)

add_executable(lab7
  main.cpp
  sdl.cpp
  document.cpp
)

add_subdirectory(lib/SDL2/)
target_link_libraries(lab7 SDL2-static)
target_include_directories(lab7 PRIVATE ${SDL2_INCLUDE_DIR})

add_subdirectory(lib/imgui/)
target_include_directories(imgui PRIVATE lib/SDL2/include/)
target_link_libraries(lab7 imgui)
```

3. **Ссылка на репозиторий:**
   https://github.com/GitGood2000/oop_exercise_07

4. **Набор testcases:**

test.txt

```
square
358 60
311 167
418 214
465 107
255 0 0
rectangle
591 67
```

682 87
640 280
549 260
0 255 0
trapezoid
330 469
678 473
582 350
430 350
0 0 255

test2.txt
triangle
392 46
304 123
367 154
255 0 0
square
506 68
512 133
577 127
571 62
0 255 0
rectangle
687 45
620 137
695 191
762 99
0 0 255
trapezoid
344 274
461 269
418 208
383 209
255 255 0
polyline 14
581 205
686 256
764 363
729 483
560 539
423 530
365 460
350 350
306 292
336 190
427 155
437 83
503 24
560 21
0 255 255
polygon 7
632 262
700 309
715 372
682 379
598 323
565 263
632 262
255 0 255
circle
483 399 99.4636
255 255 255

test3.txt

```
circle
375 291 85.7263
255 255 255
circle
601 294 96.6747
255 255 255
square
339 253
409 254
408 324
338 323
0 0 255
square
562 291
621 244
668 303
609 350
0 0 255
triangle
464 385
421 458
526 464
255 0 0
polyline 8
298 491
357 523
424 539
503 546
571 537
622 501
682 452
707 405
255 0 0
trapezoid
307 200
443 177
404 133
330 144
0 255 255
trapezoid
574 154
705 206
690 150
625 126
0 255 255
rectangle
304 90
294 23
449 0
459 67
0 255 0
rectangle
487 62
489 8
627 10
625 64
0 255 0
rectangle
647 74
659 31
734 48
722 91
0 255 0
```

polygon 6
31 537
40 585
129 587
212 565
212 552
31 537
255 255 0
polygon 6
34 404
40 344
51 279
40 196
19 191
34 404
255 0 255

5. **Результаты выполнения тестов:**

Все фигуры из тестовых файлов успешно загружены и нарисованы правильно

6. **Объяснение результатов работы программы:**
   1) Создаётся чёрный экран - “Холст“
   2) Программа выполняет определённые действия в зависимости от нажатой кнопки:
      A) “New canvas“ - стирает все фигуры;
      B) “Save\Open“ - Сохраняет координаты вершин фигур в файл или создаёт фигуры по координатам из файла;
      C) “RGB“ - Изменение цвета линий для следующей нарисованной фигуры;
      D) “Triangle/Square/Rectangle/Trapezoid/Poly-Line/Polygon/Circle“ - Создаёт фигуру (Треугольник(базовое было дано вместе с GUI, Квадрат, Прямоугольник, Трапецию, Случайный Многоугольник, Ломаную Линию, Круг)), рисует её и добавляет её в вектор
      E) “Remove“ - Удаление фигуры по индексу
      F) “Undo“ - Отменяет последнее совершенное действие (Добавление или Удаление фигуры)

7. **Вывод:** 1) Ознакомились с проектированием и структурами классов в C++.и усвоили навык работы с ними; 2) Написана программа, производящая операции на графическом интерфейсе.