МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВЦИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЬЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №8

по курсу объектно-ориентированное программирование I семестр, 2019/20 уч. год

Студент *Попов Данила Андреевич, группа М8О-208Б-18*
Преподаватель *Журавлёв Андрей Андреевич*

Условие

Работа с ассинхронностью.

Редактор должен соответствовать следующему функционалу:

1. размер буфера должен задаваться через командную строку
2. результат обработки буфера должен выводиться на экран и в файл
3. в программе должно быть два потока
4. должен прослеживаться паттерн publish-subscribe

Описание программы

Исходный код лежит в 12 файлах:

1. app/main.cpp
2. include/async.hpp
3. include/point.hpp
4. include/polygon.hpp
5. include/publisher.hpp
6. include/serializable.hpp
7. include/subscriber.hpp
8. src/async.cpp
9. src/publisher.cpp
10. src/serializable.cpp
11. src/subscriber.cpp

Дневник отладки

Race condition при инициализации второго потока.

Недочёты

На одну структуру приходится два вложенных shared$_ptr$.

Выводы

Странная лабораторная работа с мультипоточностью, в которой одновременно может выполняться только один поток. Устранил неочевидный race condition. Между делом разобрался с SFINAE. А так же ужасные отчёты, требующие много текста в выводах.

Исходный код

main.cpp

```
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <string>
#include <random>
#include <algorithm>

#include <publisher.hpp>
#include <subscriber.hpp>
#include <point.hpp>
#include <polygon.hpp>

auto constexpr default_limit = 3;

static char g_chars[] =
    "0123456789"
    "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

using rhombus  = basic_polygon<point2d, 4>;
using pentagon = basic_polygon<point2d, 5>;
using hexagon  = basic_polygon<point2d, 6>;

struct my_event final
    : oop::event {
    explicit my_event(std::shared_ptr<oop::serializable> s)
        : serializable(std::move(s))
    {}

    std::shared_ptr<oop::serializable> serializable;
};

struct unique_file_writer final
    : oop::subscriber {

    unique_file_writer()
        : rng_(std::random_device{}())
        , dist_(0, sizeof g_chars - 2)
        , unique_(unique_string_len, '\0') {
```

```
        const auto generator = [&]() {
            return g_chars[dist_(rng_)];
        };
        std::generate_n(unique_.begin(), unique_string_len, generator);
    }

    void new_unique_file() {
        if (file_.is_open()) {
            file_.close();
        }

        const auto name = generate_unique_name();
        file_.open(name, std::ios_base::out);

        ++file_counter_;
    }

    [[nodiscard]] std::string_view get_unique() const {
        return unique_;
    }

private:
    static auto constexpr unique_string_len = 16;

    size_t                          file_counter_ = 0;
    std::ofstream                   file_;
    std::default_random_engine      rng_;
    std::uniform_int_distribution<> dist_;
    std::string                     unique_;

    [[nodiscard]] std::string generate_unique_name() const {
        const auto prefix    = "./out-";
        const auto suffix    = "-";
        const auto postfix   = ".txt";
        const auto ix        = std::to_string(file_counter_);

        std::string unique(unique_string_len, '\0');

        return prefix + unique_ + suffix + ix + postfix;
    }

    void handle(const oop::event& e) override {
```

```
        if (!file_.is_open()) {
            throw std::logic_error("unique_file_writer: unique file is not generated");
        }

        const auto& my_e = dynamic_cast<const my_event&>(e);
        my_e.serializable->write(file_);
    }
};

struct stream_writer final
    : oop::subscriber {
    explicit stream_writer(std::ostream& stream)
        : stream(stream)
    {}

    std::ostream& stream;

private:
    void handle(const oop::event& e) override {
        const auto& my_e = dynamic_cast<const my_event&>(e);
        my_e.serializable->write(stream);
    }
};

size_t parse_limit(int argc, char* argv[]);
void read_rhombus(std::istream& in, rhombus& r);

int main(const int argc, char* argv[]) {
    auto const limit = parse_limit(argc, argv);
    if(!limit) {
        std::cout << "Error: Can't parse limit value." << std::endl;
        return 1;
    }

    oop::publisher     publisher;
    stream_writer      sw(std::cout);
    unique_file_writer fw;
    size_t             count = 0;

    std::cout << "Unique name: " << fw.get_unique() << std::endl;

    publisher.subscribe(&sw);
```

```
publisher.subscribe(&fw);

std::string command;
while(std::cin >> command) {
    if (command == "e" || command == "exit") {
        if (count != 0) {
            std::cout << "You can't exit till have uncommitted figures.\n"
                         "Type `force' to commit immediately." << std::endl;
            continue;
        }
        break;
    }

    bool force = false;
    if (command == "force") {
        if (count == 0) {
            std::cout << "Nothing to commit." << std::endl;
            continue;
        }
        force = true;
    }
    else {
        std::shared_ptr<oop::serializable> fig;
        if (command == "rhombus") {
            auto r = new rhombus;
            read_rhombus(std::cin, *r);
            fig.reset(r);
        }
        else if (command == "pentagon") {
            fig.reset(new pentagon{ std::cin });
        }
        else if (command == "hexagon") {
            fig.reset(new hexagon{ std::cin });
        }
        else {
            std::cout << "Unknown figure type or command." << std::endl;
            continue;
        }
        std::shared_ptr<const oop::event> e{ new my_event(fig) };
        publisher.push(e);
        ++count;
    }
```

```
        if (count == limit || force) {
            fw.new_unique_file();
            publisher.commit();
            count = 0;
        }
    }
}

size_t parse_limit(const int argc, char* argv[]) {
    auto constexpr error_occured = 0;

    if (argc == 1) {
        return default_limit;
    }
    if (argc > 2) {
        return error_occured;
    }

    char* end;
    auto const lim = std::strtoull(argv[1], &end, 10);
    if (end == nullptr) {
        return error_occured;
    }

    return lim;
}

void read_rhombus(std::istream& in, rhombus& r) {
    auto constexpr precision = 0.000000001L;
    for (auto& p : r) {
        in >> p;
    }
    if (in.fail()) {
        return;
    }

    constexpr size_t size = rhombus::size();
    const double dist     = distance(r[0], r[size - 1]);
    for (size_t i = 0; i < size - 1; i++) {
        const double next = distance(r[i], r[i + 1]);
        if (std::abs(dist - next) > precision) {
```

```
            in.setstate(std::ios::failbit);
            break;
        }
    }
}
```

include/async.hpp

```cpp
#pragma once

namespace oop {
    struct event {
        event()                             = default;
        event(const event&)                 = default;
        event(event&&) noexcept             = default;
        event& operator=(const event&)      = default;
        event& operator=(event&&) noexcept = default;

        virtual ~event() = 0;
    };
}
```

src/async.cpp

```cpp
#include "async.hpp"

oop::event::~event() = default;
```

include/point.hpp

```cpp
#pragma once

#include <iostream>
#include <cstddef>
#include <cmath>

template <typename _Type, size_t _Dimensions>
struct point {
    static_assert(_Dimensions != 0, "can not create 0d point");

    using value_type = _Type;
    using reference = value_type&;
    using const_reference = const value_type&;
    using pointer = value_type*;
    using const_pointer = const value_type*;

    using iterator = pointer;
    using const_iterator = const_pointer;

    value_type dots[_Dimensions];

    [[nodiscard]] value_type& operator[](size_t ix) noexcept {
        return dots[ix];
    }

    [[nodiscard]] const value_type& operator[](size_t ix) const noexcept {
        return const_cast<point&>(*this).operator[](ix);
    }

    [[nodiscard]] iterator begin() noexcept {
        return &dots[0];
    }

    [[nodiscard]] const_iterator begin() const noexcept {
        return const_cast<point&>(*this).begin();
    }

    [[nodiscard]] iterator end() noexcept {
        return &dots[_Dimensions];
    }
```

```
    [[nodiscard]] const_iterator end() const noexcept {
        return const_cast<point&>(*this).end();
    }

    [[nodiscard]] static constexpr size_t size() noexcept {
        return _Dimensions;
    }

    [[nodiscard]] point operator+(const point& other) const {
        point result = *this;

        for (size_t i = 0; i < result.size(); i++) {
            result[i] += other[i];
        }

        return result;
    }

    [[nodiscard]] point operator-(const point& other) const {
        point result = *this;

        for (size_t i = 0; i < result.size(); i++) {
            result[i] -= other[i];
        }

        return result;
    }
};

template <typename Type, size_t _Dims>
std::ostream& operator<<(std::ostream& stream, const point<Type, _Dims>& p) {
    stream << "{ ";
    for (const auto& d : p) {
        stream << d << " ";
    }
    stream << "}";

    return stream;
}

template <typename _Type, size_t _Dims>
std::istream& operator>>(std::istream& stream, point<_Type, _Dims>& p) {
```

```
    for (auto& d : p) {
        stream >> d;
    }

    return stream;
}

// Examples:
using point2d = point<double, 2>;

inline double distance(const point2d& left, const point2d& right) {
    const double x = left[0] - right[0];
    const double y = left[1] - right[1];
    return std::sqrt((x * x) + (y * y));
}
```

include/polygon.hpp

```cpp
#pragma once

#include <cstddef> // size_t
#include <tuple>
#include <type_traits>
#include <istream>
#include <ostream>
#include <stdexcept>

#include <serializable.hpp>

template<typename _T>
auto print2d(std::ostream& stream, const _T& tuple);

/*
    basic_polygon traits
*/
template<typename _Vertex>
struct basic_polygon_traits {
    using vertex          = _Vertex;
    using pointer         = vertex*;
    using const_pointer   = const vertex*;
    using reference       = vertex&;
    using const_reference = const vertex&;

    using iterator       = pointer;
    using const_iterator = const_pointer;
};

/*
    basic_polygon class
    tuple-like
    structured binding is available
*/
template<typename _Vertex, size_t _NumOfPoints>
class basic_polygon
    : public oop::serializable {
    static_assert(_NumOfPoints >= 3, "can not create polygon from points when there are
    using traits = basic_polygon_traits<_Vertex>;

public:
```

```
using vertex          = typename traits::vertex;
using pointer         = typename traits::pointer;
using const_pointer   = typename traits::const_pointer;
using reference       = typename traits::reference;
using const_reference = typename traits::const_reference;

using iterator       = typename traits::iterator;
using const_iterator = typename traits::const_iterator;


// constructors
basic_polygon() = default;
explicit basic_polygon(std::istream& stream) {
    for (auto& point : points) {
        stream >> point;
    }
    if (stream.fail()) {
        throw std::runtime_error("bad polygon initialization");
    }
}
explicit basic_polygon(const vertex& v) noexcept {
    for (auto& point : points) {
        point = v;
    }
}



// element getters
reference at(size_t ix) {
    return points[ix];
}
const_reference at(size_t ix) const {
    return const_cast<basic_polygon&>(*this).at(ix);
}

reference operator[](size_t ix) {
    return at(ix);
}
const_reference operator[](size_t ix) const {
    return const_cast<basic_polygon&>(*this)[ix];
}
```

```
// iterators
iterator begin() {
    return &points[0];
}
const_iterator begin() const {
    // cast const to mutable and use non-const begin
    return const_cast<basic_polygon&>(*this).begin();
}

/* NEVER DEREFERENCE */
iterator end() {
    return &points[_NumOfPoints];
}
/* NEVER DEREFERENCE */
const_iterator end() const {
    // cast const to mutable and use non-const end
    return const_cast<basic_polygon&>(*this).end();
};



// structured binding
template<size_t _Ix>
constexpr auto& get() & {
    // check out of bounds
    if constexpr (_Ix < _NumOfPoints) {
        return points[_Ix];
    }
    else {
        // generate compile-time error
        static_assert(_Ix < _NumOfPoints, "ix is out of range");
    }
}

template<size_t _Ix>
constexpr auto const& get() const& {
    // cast const to mutable and use non-const get
    // which does no effect on storage
    return const_cast<basic_polygon&>(*this).get<_Ix>();
```

```
    }

    template<size_t _Ix>
    constexpr auto&& get() && {
        // cast lvalue reference to rvalue and return it
        return std::move(this->get<_Ix>());
    }

    static constexpr size_t size() {
        return _NumOfPoints;
    }

    void write(std::ostream& s) override;

private:
    vertex points[_NumOfPoints];

    template<size_t _Ix, typename _V, size_t _N>
    friend constexpr auto std::get(const basic_polygon<_V, _N>& polygon);
};

// std types specializations for structured binding of basic_polygon
namespace std {
    template<size_t _Ix, typename _Vertex, size_t _NumOfPoints>
    constexpr auto get(const basic_polygon<_Vertex, _NumOfPoints>& polygon) {
        return polygon.points[_Ix];
    }

    template<typename _Vertex, size_t _NumOfPoints>
    struct tuple_size<::basic_polygon<_Vertex, _NumOfPoints>>
        : integral_constant<size_t, _NumOfPoints> {};

    template<size_t _Ix, typename _Vertex, size_t _NumOfPoints>
    struct tuple_element<_Ix, ::basic_polygon<_Vertex, _NumOfPoints>> {
        using type = typename basic_polygon_traits<_Vertex>::vertex;
    };
} // namespace std

template <typename _Vertex, size_t _NumOfPoints>
void basic_polygon<_Vertex, _NumOfPoints>::write(std::ostream& s) {
    print2d(s, *this);
}
```

```
namespace detail {
    template<size_t _Off, size_t ... _Ix>
    std::index_sequence<(_Off + _Ix)...> add_offset(std::index_sequence<_Ix...>) {
        return {};
    }

    template<size_t _Off, size_t _N>
    auto make_index_sequence_with_offset() {
        return add_offset<_Off>(std::make_index_sequence<_N>{});
    }

    template<typename _T, size_t... _Ix>
    double area2d(const _T& tuple, std::index_sequence<_Ix...>) {
        using vertex = std::remove_const_t<std::remove_reference_t<decltype(std::get<0>(
        static_assert(std::is_same_v<vertex, point2d>, "incorrect type");

        auto constexpr tuple_size = std::tuple_size<_T>{}();
        auto constexpr x = 0;
        auto constexpr y = 1;

        using std::get;

        double result = ((get<_Ix>(tuple)[x] * (get<_Ix + 1>(tuple)[y] - get<_Ix - 1>(tu
        auto constexpr first = 0;
        auto constexpr last = tuple_size - 1;
        result += get<first>(tuple)[x] * (get<first + 1>(tuple)[y] - get<last>(tuple)[y]
        result += get<last>(tuple)[x] * (get<first>(tuple)[y] - get<last - 1>(tuple)[y])
        result /= 2;

        return std::abs(result);
    }

    template<typename _T, std::size_t... _Ix>
    auto center2d(const _T& tuple, std::index_sequence<_Ix...>) {
        using vertex = std::remove_const_t<std::remove_reference_t<decltype(std::get<0>(
        static_assert(std::is_same_v<vertex, point2d>, "incorrect type");

        auto constexpr tuple_size = std::tuple_size<_T>{}();
        auto constexpr x = 0;
        auto constexpr y = 1;
```

```
        vertex result = (std::get<_Ix>(tuple) + ...);
        result[x] /= tuple_size;
        result[y] /= tuple_size;

        return result;
    }

    template<typename _T, std::size_t... _Ix>
    auto print_points2d(std::ostream& out, const _T& tuple, std::index_sequence<_Ix...>)
        auto constexpr tuple_size = std::tuple_size<_T>{}();
        (out << ... << std::get<_Ix>(tuple));
    }
}

template<typename _T>
double area2d(const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    using vertex = std::remove_reference_t<decltype(std::get<0>(tuple))>;
    return detail::area2d(tuple, detail::make_index_sequence_with_offset<1, tuple_size -
}

template<typename _T>
auto center2d(const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();
    return detail::center2d(tuple, std::make_index_sequence<tuple_size>{});
}

template<typename _T>
auto print2d(std::ostream& stream, const _T& tuple) {
    auto constexpr tuple_size = std::tuple_size<_T>{}();

    using std::endl;

    stream << "\ntype:   ";
    switch (tuple_size) {
    case 4:
        stream << "rhombus" << endl; break;
    case 5:
        stream << "pentagon" << endl; break;
    case 6:
        stream << "hexagon" << endl; break;
    default:
```

```
        stream << "unknown" << endl;
    }

    stream << "center: " << center2d(tuple) << endl
        << "area:   " << area2d(tuple) << endl
        << "points: ";
    detail::print_points2d(stream, tuple, std::make_index_sequence<tuple_size>{});
    stream << endl << endl;
}
```

include/publisher.hpp

```cpp
#pragma once

#include <vector>
#include <list>
#include <memory>
#include <mutex>
#include <condition_variable>
#include <thread>

#include <async.hpp>

namespace oop {
    class subscriber;

    class publisher final {
    public:
        publisher();
        ~publisher();

        publisher(const publisher&)                = delete;
        publisher(publisher&&) noexcept            = delete;
        publisher& operator=(const publisher&)     = delete;
        publisher& operator=(publisher&&) noexcept = delete;

        /*!
         * @brief Push next event.
         *
         * @param e
         * pointer to const event
         */
        void push(const std::shared_ptr<const event>& e);

        /*!
         * @brief Commit current events queue.
         *
         * Function does NOT RETURN till committing is not complete.
         */
        void commit();

        /*!
         * @brief Add new subscriber.
```

```
         *
         * @param s
         * pointer to new subscriber
         */
        void subscribe(subscriber* s);

    private:
        std::mutex              publisher_mu_;
        std::condition_variable publisher_cv_;

        std::vector<std::shared_ptr<const event>> events_;

        std::list<subscriber*>    subscribers_;
        std::thread               routine_;
        std::mutex                routine_mu_;
        std::condition_variable   routine_cv_;
        bool                      events_done_;

        void routine_proc();
        void stop_routine();
    };
}
```

src/publisher.cpp

include/serializable.hpp

```cpp
#pragma once

#include <ostream>

namespace oop {
    struct serializable {
        serializable()                                  = default;
        serializable(const serializable&)               = default;
        serializable(serializable&&) noexcept           = default;
        serializable& operator=(const serializable&)    = default;
        serializable& operator=(serializable&&) noexcept = default;

        virtual ~serializable()             = 0;
        virtual void write(std::ostream& s) = 0;
    };
}
```

src/serializable.cpp

```cpp
#include "serializable.hpp"

oop::serializable::~serializable() = default;
```

include/subscriber.hpp

```cpp
#pragma once

#include <async.hpp>

namespace oop {
    class subscriber {
    public:
        subscriber()                                    = default;
        subscriber(const subscriber&)                   = default;
        subscriber(subscriber&&) noexcept               = default;
        subscriber& operator=(const subscriber&)        = default;
        subscriber& operator=(subscriber&&) noexcept = default;

        virtual ~subscriber() = 0;

    private:
        virtual bool is_suitable(const event& e) { return true; }
        virtual void handle(const event& e) = 0;

        friend class publisher;
    };
}
```

src/subscriber.cpp

```cpp
#include "subscriber.hpp"

oop::subscriber::~subscriber() = default;
```