Writing and reading objects

```
/*
      . Topics:
          . Serilizing POD (Plain Old Data) types
          . Serializing non-POD types
          . ...


*/
```

Writing and reading objects

```
export struct PODType {
    int id;
    double value;
};

// Function to write POD to a binary file (returns bool for success)
export bool write_pod(const std::filesystem::path& filename, const PODType& pod) {
    std::ofstream out_file(filename, std::ios::binary);
    if (!out_file) {
        return false;
    }
    out_file.write(reinterpret_cast<const char*>(&pod), sizeof(PODType));
    return out_file.good();
}

// Function to read POD from a binary file (returns bool for success)
export bool read_pod(const std::filesystem::path& filename, PODType& pod) {
    std::ifstream in_file(filename, std::ios::binary);
    if (!in_file) {
        return false;
    }
    in_file.read(reinterpret_cast<char*>(&pod), sizeof(PODType));
    return in_file.good();
}
```

Writing and reading several objects

```cpp
// Function to write a vector of PODType objects to a file
export bool write_pod_vector(const std::filesystem::path& file_path,
                             const std::vector<pod_types::PODType>& pod_vector) {
    std::ofstream out_file(file_path, std::ios::binary);
    if (!out_file) {
        fmt::print("Failed to open file for writing.\n");
        return false;
    }

    for (const auto& pod : pod_vector) {
        out_file.write(reinterpret_cast<const char*>(&pod), sizeof(pod_types::PODType));
    }

    if (!out_file.good()) {
        fmt::print("Failed to write PODType objects to file.\n");
        return false;
    }

    out_file.close();
    return true;
}
```

Writing and reading several objects

```
// Function to read a vector of PODType objects from a file
export bool read_pod_vector(const std::filesystem::path& file_path,
                            std::vector<pod_types::PODType>& pod_vector) {
    std::ifstream in_file(file_path, std::ios::binary);
    if (!in_file) {
        fmt::print("Failed to open file for reading.\n");
        return false;
    }

    pod_types::PODType read_pod;
    while (in_file.read(reinterpret_cast<char*>(&read_pod), sizeof(pod_types::PODType))) {
        pod_vector.push_back(read_pod);
    }

    if (!in_file.eof() && !in_file.good()) {
        fmt::print("Failed to read PODType objects from file.\n");
        return false;
    }

    in_file.close();
    return true;
}
```

Writing and reading 1 object

```
export void read_write_pod_types(){

    // Define file path
    std::filesystem::path file_path = R"(D:\sample_pod_file.bin)"; // Windows
    //std::filesystem::path file_path = R"(/path/to/your/input_file.bin)"; // Linux

    // Create a PODType object
    pod_types::PODType pod = {42, 3.14159};

    // Write the PODType object to a binary file
    if (pod_types::write_pod(file_path, pod)) {
        fmt::print("PODType object written to file.\n");
    } else {
        fmt::print("Failed to write PODType object to file.\n");
    }

    // Read the PODType object from the binary file
    pod_types::PODType read_pod;
    if (pod_types::read_pod(file_path, read_pod)) {
        fmt::print("PODType object read from file: id={}, value={}\n", read_pod.id, read_pod.value);
    } else {
        fmt::print("Failed to read PODType object from file.\n");
    }

}
```

Writing and reading several objects

```
export void write_and_read_several_pod_types(){

    // Define file path
    std::filesystem::path file_path = R"(D:\sample_pod_vector.bin)"; // Windows
    //std::filesystem::path file_path = R"(/path/to/your/input_file.bin)"; // Linux

    // Create a vector of PODType objects
    std::vector<pod_types::PODType> pod_vector = {
        {1, 1.1}, {2, 2.2}, {3, 3.3},
        {4, 4.4}, {5, 5.5}
    };

    // Write the vector of PODType objects to a binary file
    if (pod_types::write_pod_vector(file_path, pod_vector)) {
        fmt::print("PODType objects written to file.\n");
    } else {
        fmt::print("Failed to write PODType objects to file.\n");
    }

    // Read the vector of PODType objects from the binary file
    std::vector<pod_types::PODType> read_pod_vector;
    if (pod_types::read_pod_vector(file_path, read_pod_vector)) {
        fmt::print("PODType objects read from file:\n");
        for (const auto& pod : read_pod_vector) {
            fmt::print("id={}, value={}\n", pod.id, pod.value);
        }
    } else {
        fmt::print("Failed to read PODType objects from file.\n");
    }
}
```

Non POD Types

```cpp
export class Person {
public:
    std::string name;
    int age;

    // Serialization function for a single person
    void serialize(std::ofstream& ofs) const {
        size_t name_size = name.size();
        ofs.write(reinterpret_cast<const char*>(&name_size), sizeof(name_size));
        ofs.write(name.data(), name_size);
        ofs.write(reinterpret_cast<const char*>(&age), sizeof(age));
    }

    // Deserialization function for a single person
    void deserialize(std::ifstream& ifs) {
        size_t name_size;
        ifs.read(reinterpret_cast<char*>(&name_size), sizeof(name_size));

        name.resize(name_size);

        //Read the name and age from the file and store in the object's members: name and age
        ifs.read(name.data(), name_size);
        ifs.read(reinterpret_cast<char*>(&age), sizeof(age));
    }
};
```

Non POD Types

```
// Function to write a vector of person objects to a file
export void write_persons_to_file(const std::vector<Person>& persons, const std::filesystem::path& file_path) {
    std::ofstream ofs(file_path, std::ios::binary);
    if (!ofs) {
        fmt::println("Failed to open file for writing: {}", file_path.string());
        return;
    }

    size_t vector_size = persons.size();
    ofs.write(reinterpret_cast<const char*>(&vector_size), sizeof(vector_size));

    for (const auto& p : persons) {
        p.serialize(ofs);
    }

    fmt::println("Serialized {} person(s) to file: {}", vector_size, file_path.string());
}
```

Non POD Types

```cpp
// Function to read a vector of person objects from a file
export std::vector<Person> read_persons_from_file(const std::filesystem::path& file_path) {
    std::ifstream ifs(file_path, std::ios::binary);
    if (!ifs) {
        fmt::println("Failed to open file for reading: {}", file_path.string());
        return {};
    }

    size_t vector_size;
    ifs.read(reinterpret_cast<char*>(&vector_size), sizeof(vector_size));

    std::vector<Person> persons(vector_size);
    for (auto& p : persons) {
        p.deserialize(ifs);
    }

    fmt::println("Deserialized {} person(s) from file: {}", vector_size, file_path.string());
    return persons;
}
```

Non POD Types

```
export void write_read_several_non_pod_types(){

    std::vector<non_pod_types::Person> persons = {{"Alice", 30}, {"Bob", 25}, {"Charlie", 40}};
    std::filesystem::path file_path = R"(D:\sample_non_pod_vector.bin)"; // Windows
    //std::filesystem::path file_path = R"(/path/to/your/input_file.bin)"; // Linux


    // Serialize the vector of person objects
    non_pod_types::write_persons_to_file(persons, file_path);

    // Deserialize the vector of person objects
    std::vector<non_pod_types::Person> loaded_persons = non_pod_types::read_persons_from_file(file_path);

    // Print the loaded data
    for (const auto& p : loaded_persons) {
        fmt::println("Name: {}, Age: {}", p.name, p.age);
    }
}
```

Serializing Pixel

```
// Pixel class definition
 export class Pixel {
    public:
    // Constructors


    // Serialization: Save a Pixel object to a binary file
    void serialize(std::ofstream& out) const {
        out.write(reinterpret_cast<const char*>(&m_color), sizeof(m_color));
        out.write(reinterpret_cast<const char*>(m_pos), sizeof(Position));
    }

    // Deserialization: Load a Pixel object from a binary file
    void deserialize(std::ifstream& in) {
        in.read(reinterpret_cast<char*>(&m_color), sizeof(m_color));
        in.read(reinterpret_cast<char*>(m_pos), sizeof(Position));
    }

    private:
        uint32_t m_color{0xFF000000};
        Position* m_pos{nullptr}; // Raw pointer for position
};
```

Serializing Pixel

```cpp
export void save_pixels(const std::vector<Pixel>& pixels, const std::filesystem::path& filepath) {
    std::ofstream out(filepath, std::ios::binary);
    if (!out) {
        fmt::println("Error opening file for writing: {}\n", filepath.string());
        return;
    }
    for (const auto& pixel : pixels) {
        pixel.serialize(out);
    }
    out.close();
}

export std::vector<Pixel> load_pixels(const std::filesystem::path& filepath) {
    std::vector<Pixel> pixels;
    std::ifstream in(filepath, std::ios::binary);
    if (!in) {
        fmt::println("Error opening file for reading: {}\n", filepath.string());
        return pixels;
    }
    Pixel temp;
    while (in.peek() != EOF) {
        temp.deserialize(in);
        pixels.push_back(temp);
    }
    in.close();
    return pixels;
}
```

Serializing Pixel

```cpp
// Create a vector of Pixel objects
std::vector<ct16::Pixel> pixels = {
    ct16::Pixel(0xFF0000FF, 10, 20), // Red pixel at (10, 20)
    ct16::Pixel(0xFF00FF00, 30, 40), // Green pixel at (30, 40)
    ct16::Pixel(0xFFFF0000, 50, 60)  // Blue pixel at (50, 60)
};

// Define the file path for serialization
std::filesystem::path file_path = R"(D:\serialized_pixels.bin)"; // Windows
//std::filesystem::path file_path = R"(/path/to/your/input_file.bin)"; // Linux

// Serialize the vector of Pixel objects to a file
ct16::save_pixels(pixels, file_path);

//Deserialize the vector of Pixel objects from the file
std::vector<ct16::Pixel> deserialized_pixels = ct16::load_pixels(file_path);

// Print the deserialized Pixel objects to verify
fmt::println("Deserialized Pixels from binary file");
for (const auto& p : deserialized_pixels) {
    auto position = p.get_position();
    fmt::print("Pixel color: {:#08X}, Position: ({}, {})\n", p.get_color(), position.x, position.y);
}
```