Object-Serialization

2024-ZJU-OOP Final-Project cx

Project completion status

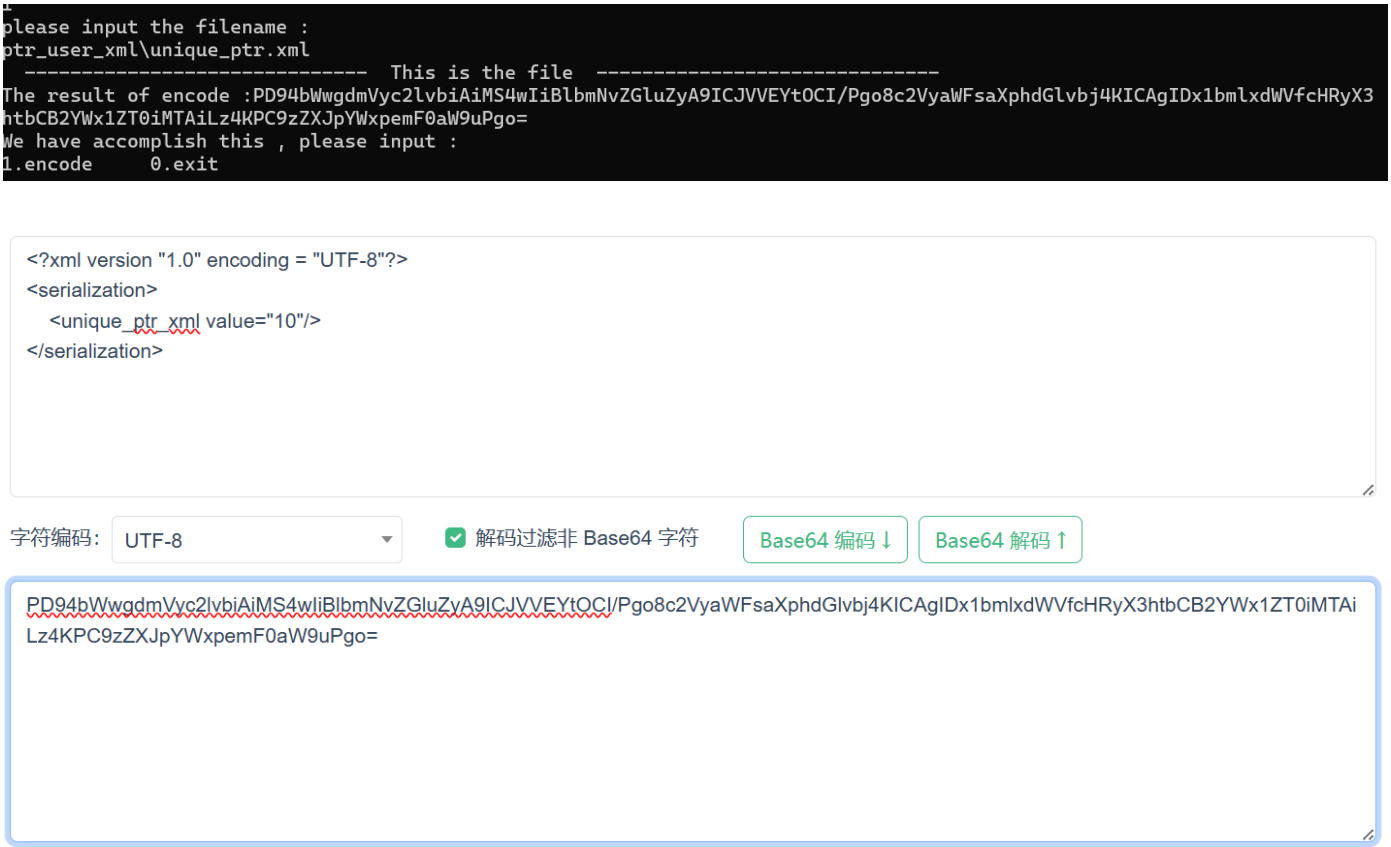
Note: **this project is built on windows**

I have written makefile files for each module, and during testing, I first perform mingw32-make clean, and then use mingw32-make to implement compilation. Finally run \binary.exe or \xml.exe.

> For binary serialization/deserialization , I use template. For some implementations that use base64 encryption and decryption.

> Here, for xml , I generated relevant files for each type (std:: pair, std:: vector, std:: list, and unique_ptr) and encrypted them using base64.

Meanwhile, when implementing smart pointers, we used related functions such as. lock() and. expired(), due to their different characteristics



This can prove that our base64 result is correct. **and it should be noted that when filling in our file path, we follow the following format:**

- ptr_user_xml\unique_ptr.xml
- xml_generate\int.xml

In use, I used c++20

Explaination

In binary_test, we have implemented all the required functionalities and have simply implemented the calculation of base64 for some. data files through macros.

Among them, binary.cpp includes all the tests, binary_fun.cpp is the specific function implementation, and binary.hpp has related function definitions, macro definitions, as well as template and namespace definitions.

Include_file. h is used to include all the header files we need.

For xml_test, the file framework is similar, fully implementing all functions and Bonus.

Project requirements

Description

In computer science, serialization is the process of translating object state into a format that can be stored/transmitted and reconstructed later.

Binary serialization uses binary encoding to produce a compact result for uses such as storage or socket-based network streams. This mechanism is especially useful if you have to handle big data where the loading/saving time is crucial.
For cases where you want to read/edit the serialized data, e.g., for software configuration, a serialization to text files (XML, JSON, etc.) is a natural choice.

Please refer to Serialization for more details.

Requirements

- Implement a module to support binary serialization/deserialization:

```cpp
int n0 = 256, n1;
// serialize object n0 to a binary file n.data
serialize(n0, "n.data");
// reconstruct object n1 from the content of n.data
deserialize(n1, "n.data");
// now n0 == n1 should be true.
```

- Implement a wrapper module of tinyxml2 to support **XML** serialization:

```cpp
std::pair<int, double> pair0 = {2, 3.1}, pair1;
// serialize object pair0 to an XML file pair.xml with the name std_pair
serialize_xml(pair0, "std_pair", "pair.xml");
// reconstruct object pair1 from the content of pair.xml
deserialize_xml(pair1, "std_pair", "pair.xml");
// now pair0 == pair1 should be true.
```

The pair.xml would be something like:

```xml
<serialization>
<std_pair>
    <first val="2"/>
    <second val="3.1000000000000001"/>
</std_pair>
</serialization>
```

- Every module has its namespace.
- Both modules should at least support the serialization of arithmetic types (see `std::is_arithmetic`), C++ string type (`std::string`), and STL containers (`std::pair`, `std::vector`, `std::list`, `std::set`, and `std::map`).
- Both modules should provide a convenient mechanism (by macro, template, etc.) to support the serialization of user-defined types, e.g.,
- During testing, you should cover all the required types and options.
- **Bonus** Use binary-to-text encoding/decoding (base64) to implement a binary mode of XML serialization.
- **Bonus** Support the serialization of smart pointers, e.g., `std::unique_ptr`.