MEMO

Introduction:

This is a memo describing the basic functions and implementations of our serialization API. This API is mostly used to serialize and deserialize different objects, especially network messages.

Design:

We have a serial.h file which includes methods of serializing and deserializing primitive types, such as double, size_t and int, and methods of deserialzing arrays and messages. Our serialization of arrays and messages are defined inside their own classes, this is designed to make every object capable of being serialized to create code flexibility.

The way we serialize primitive types is simply by calling reinterpret_cast<char*>() on it to create a char pointer which points to the char array representation. For deserializing, we just cast the char pointer back to a pointer to the object type. Like:

```
char* serial_double(double num) {
        char* buf = reinterpret_cast<char*>(&num);
        return buf
}
double deserial_double(char* buf) {
        return *reinterpret_cast<double*>(buf);
}
```

For serializing arrays and messages, we complete these by serializing each element inside separately and putting them together into a large char array in the end. We use some delimiters like “ “ and other char symbols to separate each element’s char* representation, so when we deserialize them, we split the char array by these delimiters and deserialize these elements one by one.

Use cases:

Our serialization API supports several functionalities:

1. Serializing primitive types of double, size_t and int.
2. Serializing self-defined arrays of double and string.
3. Serializing network messages of Ack, Status and Directory(except Register which has structure field).

Users can use our API to serialize such information by calling serialize() on the object they want to serialize(). This method will return a sequence of bytes representing the char array of that object. For example of serializing a double array:

```
double d = 8.8;
DoubleArray* da = new DoubleArray();
```

```
        da->add(d);
        da->add(d);
        char* dbuf = da->serialize();
```

Besides, users can deserialize the char pointer representation to the object itself by calling specifc deserialize method which takes in the char pointer they want to deserialize. For the previous example:

```
        DoubleArray* daa = deserial_double_array(dbuf);
        std::cout << da->equals(daa) << "\n";
```

deserial_double_array will return an array of double and in this case this array is equals to da.

If a user want to deserialize a message, he or she can simply call deserial_message(). This method deals with char representations of all kinds of messages and returns the specific kind of message with its information which the input char array refers to. For example of Ack message:

```
        Ack* ack = new Ack(1, 2, 3);
        char* ackbuf = ack->serialize();
        Message* ack1 = deserial_message(ackbuf);
        std::cout << (ack == ack1) << "\n";
```

Deserialization in this case will return a new Ack message which is exactly the same as the original Ack message.

Challenges:

The biggest challenge we faced is how to serialize enum class and struct in c++. In fact, we figured out don’t need to serialize enum class as we could add a symbol like a number in each messages’ char array representation to keep track of MsgKind, instead of serializing it using cast. However, we are struggling with struct of sockaddr_in since it has several sub fields of different types which are hard to serialize. We have a basic idea of how to do this, like serializing these sub fields each and putting them in a specific place inside the char representation, but we don’t have enough time to implement that.