Serialization Overview

Serialization is the process of converting a data structure or object into a format that can be easily stored in a file or data buffer as a sequence of bytes. In this assignment, we will focus designing a program that can read from and write to a file, but the same concept applies directly to sending packets over the internet. There are two main types of serialization methods: **text** and **binary**. A *text serializer* converts an object into text (e.g., an `std::string` in C++). For instance, say we have a simple `Point` struct[1] as follows:

```
struct Point {
 int x;
 int y;
};

Point p{.x = -23421, .y = 758392};
```

A text serializer for `p` could produce the following output: "`{x:-23421, y:758392}`" in a format similar to a Python dictionary or JSON. Here, the output consists of 20 characters, which translates to 20 bytes as each character is 1 byte[2]. The benefits of a text-based serializer are that the output is human-readable and does not rely on the underlying hardware (e.g., 32-bit versus 64-bit processors). The main downside is that the representation is less space efficient than binary serializers. For instance, in the above example, we know that `p` is only 8 bytes as both `p.x` and `p.y` are 4-byte integers. This means that the 20-byte text representation uses $\frac{20}{8} = 2.5\times$ more space in this small example. Extrapolated to larger objects (e.g., on the order of megabytes) this $2.5\times$ can be considered a significant overhead. In contrast, a *binary serializer* for `p` could produce an 8 byte output corresponding to the underlying integer representation of `p`. For those curious, on my system the binary representation is: `11111111 11111111 10100100 10000011 11111111 11111111 10100100 10000011`.

We are also interested in deserialization, which is the process of converting a sequence of characters/bytes into an object. After all, we want to use these serialized formats at some point in the future! For instance, we could write a `text_deserializer` function that takes a `std::string` corresponding to a serialized text format and returns a `Point` that is initialized with the string. Below is an code example of this process:

```
std::string serialized_point = "{x:-23421, y:758392}";
Point p = text_deserializer(serialized_point);
std::cout << p.x << ", " << p.y << std::endl; //Prints -23421, 758392
```

[1] I generally dislike using classes for simple data types - but the same notion applies to them as well.

[2] We will assume ASCII in this class. In general, modern applications use Unicode with variable bytes per symbol.

Writing a Custom Serializer and Deserializer

For this assignment, you will be writing serializers and deserializers for a custom data format used by a fictitious bakery to track their orders. The text format for these orders is as follows:

```
@employees
Brad
Claudia
Simone

@items
Biscuit, 3.50
Bun, 0.99
Brownie, 4.75
White Loaf, 7.50
Wheat Loaf, 8.25

@orders
Claudia: 4 Wheat Loaf, 7 Biscuit, 6 Bun, 4 Brownie
Brad: 1 Bun, 2 Brownie, 8 Biscuit, 1 White Loaf
Brad: 8 Brownie, 4 Bun, 5 Wheat Loaf
```

As you see, the file has three sections: `@employees`, `@items`, and `@orders`. Each item has a name and a price. Each order has an employees tied to it at the beginning of the line, followed by a list of items in the order (with quantities). For instance, the first order has 4 Wheat Loafs and the third order has 5 Wheat Loafs. You are allowed to make several assumptions about the contents of the file:

- No error handling is required - it will always be formatted exactly like this.
- It is impossible for an order to contain an item that is not in the items portion of the file.
- Likewise, an employee in the employees section must be used for each order.
- The bakery caps each quantity to 9 per item and each item will only appear once per order.

To help get started, I have provided a github repository with more instructions on how to build and run the C++ code. We will be using the CMake build system for all assignments in class. I have set it up for this project so that it should be minimal work for you (you can learn more about CMake as you go). I have also provided an implementation of the text serializer for free.

Your task for this assignment is to implement 3 functions in `src/bakery.cpp`: a text deserializer, a binary serializer, and a binary deserializer. I have provided 3 bakery order files of varying sizes (`data/small.txt`, `data/medium.txt`, and `data/large.txt`). For `data/large.txt`, the text format is 3.9MB. To receive full credit, your binary format for this file must be less than 2MB.

The other modification you should make is to `src/main.cpp` to perform both text and binary serialization and deserialization while timing each function using `std::chrono::high_resolution_clock` and report the resulting wallclock time for each operation in milliseconds (ms).

Feel free to make extensive use of google to research how to read/write bytes to/from a binary file to perform the serialization/deserialization. However, you are not allowed to take an off-the-shelf serializer, such as boost.serialization, to do the work for you. Feel free to make extensive use of the C++ standard library.

To summarize the work:

- In `bakery.cpp`, implement text deserialization
- In `bakery.cpp`, implement binary serialization
- In `bakery.cpp`, implement binary deserialization
- In `main.cpp`, time both binary and text serialization/deserialization
  - Use `std::chrono::high_resolution_clock` to do this
  - Print out each of these times with labels for the function
- Ensure that your binary serializer produces a less than 2MB file for `data/large.txt`
- Make free use of the C++ standard library but not any additional libraries

When you are finished, zip up your project (excluding the `build/` directory) into `assignment1.zip` and submit via Canvas.