# Minix CPP Proficiency Test

Design and implement a packet parser that processes data adhering to the provided input specification and generates an output file containing events formatted as per the given output specification. The output file should include events written sequentially, one after another, without any gaps. For example, if the output contains two "Add Order" messages (44 bytes each), the first message should occupy bytes 0–43, and the second should occupy bytes 44–87.

The parser will be assessed based on correctness, performance (both time and space), and code readability. Additionally, you are expected to create regression or unit tests to validate your parser. Testing is a critical aspect of Minix's development process. Include a `test_plan.md` file in your submission that outlines the steps taken to test the parser. Starter code is provided, including an interface to implement and a Makefile for compiling the code into a static library. We will compile your code using `make libparser.a` and test it using our test harness. If you add regression or unit tests, document how to execute them in a `README` file.

The provided Makefile also includes a target named `feed` (buildable using `make`). This generates a binary that reads a sample input file (`test.in`), containing two packets with "Add Order" messages, and invokes `Parser::onUDPPacket()`, which currently prints a line for each packet.

The parser will be tested on a Debian Stretch system with an x86-64 processor and compiled using g++ version 11.

## Input Specification

### Data Types

- All integer fields use big-endian format.
- Alphanumeric fields are left-justified ASCII, padded with spaces.
- Prices are integers with 4 implied decimal places.
- Timestamps represent nanoseconds since midnight (local time).

### Message Format

Each UDP packet begins with a header containing:

1. **Packet Size**: Total bytes in the packet, including the header.
2. **Sequence Number**: Sequence number of the packet (no gaps expected, but packets may arrive out of order or duplicated).
3. **Payload**: Contains zero or more messages, which may span packet boundaries.

## Supported Messages

1. **Add Order**: Adds a new order to the book.
2. **Order Executed**: Indicates partial or full execution of an order.
3. **Order Cancelled**: Cancels a portion of an order.
4. **Order Replaced**: Replaces an existing order with new details (side and ticker remain unchanged).

Refer to the detailed input specification for field offsets, lengths, and types.

---

## Output Specification

- Translate each input message type into the corresponding output message:
  - Add Order → Add Order
  - Order Executed → Order Executed
  - Order Cancelled → Order Reduced
  - Order Replaced → Order Replaced

## Data Types

- Integer fields are little-endian.
- Alphanumeric fields are left-justified and padded with null (`\0`).
- Prices use 8-byte little-endian floating-point numbers.
- Timestamps represent nanoseconds since the Unix epoch.

## Message Format

Each output message includes a common header:

1. **Message Type**: Identifies the type of message.
2. **Message Size**: Total bytes in the message, including the header.

Detailed formats for each message type (Add Order, Order Executed, etc.) are provided in the output specification.

---

## Development Requirements

1. Implement the provided interface in the starter code.
2. Use the Makefile to compile your code and ensure compatibility with the `libparser.a` library.
3. Include a robust test plan (`test_plan.md`) and clear instructions for running your tests (`README`).
4. Ensure the parser correctly handles out-of-order or duplicated packets, and properly processes messages spanning multiple packets.