C++ Data Parsing Project

Introduction

The code (`parser.cpp`) provided is designed to process accelerometer data stored in a file named `sample_data.txt`. To run this code, a C++ compiler such as MinGW or a similar one is needed.

Firstly, ensure the sample data file (`sample_data.txt`) and the code file (`parser.cpp`) are placed in the same folder (e.g., a folder named "sem"). The `parser.cpp` file contains the C++ code responsible for reading the accelerometer data, parsing it, performing computations, and generating outputs.

Running the code will create two additional files: one named `output_data.txt` and the other named `valid_lines.txt`. The `output_data.txt` file contains the processed data formatted into X, Y, and Z axis values, ready for analysis. Meanwhile, the `valid_lines.txt` file contains the original lines from the sample data file that passed the checksum verification process.

The `parser.cpp` code is structured to process the accelerometer data, ensuring its validity, and providing helpful insights into the nature of the data. This introductory section will walk you through the necessary steps to set up, run the code, and understand the generated output files.

Acceleration Class Explanation

The `Acceleration` class manages accelerometer data and provides functionalities to process and analyze the readings. Here's a detailed breakdown:

- **Member Variables**:
  - `x_g`, `y_g`, `z_g`: Variables representing X, Y, and Z-axis readings in 'g' units.
  - `valid_messages`: Keeps count of valid accelerometer messages processed.
  - `invalid_checksum_messages`: Counts messages with invalid checksums.
  - `alerts_encountered`: Tracks the total number of alerts triggered.
  - `max_x`, `max_y`, `max_z`: Store the maximum values encountered for each axis.
  - `z_axis_offset_enabled`: Flag indicating if Z-axis offset is enabled.
  - `alert_triggered`: Flag monitoring the alert trigger status.
- **Member Functions**:
  - `Acceleration(float x, float y, float z)`: Constructor initializing axis values and other counts and flags.

- `enableZAxisOffset()`: Method enabling the Z-axis offset.
- `updateStatistics(float x, float y, float z)`: Updates statistics like valid message count and maximum axis values.
- `incrementInvalidChecksumCount()`: Increments the count for messages with invalid checksums.
- `checkAlert(float y)`: Monitors Y-axis readings to trigger alerts if consecutive readings are above a threshold.
- `getFormattedValues()`: Generates a formatted string with axis values and appends "[ALERT]" if an alert is triggered.

The `Acceleration` class acts as a central hub for handling accelerometer data, calculating statistics, and triggering alerts based on specific conditions.

main() Function Explanation

The main function of the program performs the following tasks:

- `std::ifstream inputFile("sample_data.txt");`: Opens a file stream `inputFile` to read data from the file named "sample_data.txt". If the file does not exist or cannot be opened, it returns an error message.
- `std::ofstream outputFile("output.txt");` and `std::ofstream validChecksumFile("val` Opens file streams `outputFile` and `validChecksumFile` to write data respectively to "output.txt" and "valid_checksum_lines.txt". If these files cannot be opened, it returns an error message.
- `std::time_t currentTime = std::time(nullptr);`, `std::tm* currentTm = std::gmtime(&currentTime);`: Retrieves the current time in Coordinated Universal Time (UTC) and stores it in the variable `currentTime`. It then converts this time into a broken-down time structure using `gmtime()`, storing it in `currentTm`.
- `int netherlandsOffset = 1;`: Specifies the offset in hours for the Netherlands timezone (CET/CEST) from UTC.
- `currentTm->tm_hour += netherlandsOffset;`, `std::mktime(currentTm);`: Adjusts the hours in the time structure (`currentTm`) by adding the specified offset for the Netherlands timezone and normalizes the time structure using `mktime()`.
- `std::stringstream ss;`, `ss << std::put_time(currentTm, "%Y-%m-%dT%H:%M:%SZ");`: Creates a string stream `ss` and formats the time data from the `currentTm` structure into a string of the format "YYYY-MM-DDTHH:MM:SSZ".
- `outputFile << ss.str() << std::endl;`, `std::cout << "Current time in Netherlands timezone: " << ss.str() << std::endl;`: Writes the formatted time to the `outputFile` and displays the current time in the Netherlands timezone on the console.

Z-axis Offset Enablement Explanation

This function reads data from a file, opens output files, retrieves and formats the current time, adjusts it to the Netherlands timezone, and writes the formatted time to an output file while displaying it on the console. This segment of code handles the process of enabling or disabling the Z-axis offset in the application:

- `std::string line;` and `Acceleration acc(0, 0, 0);`: Declares a string variable `line` and an instance of the `Acceleration` class named `acc`, initializing it with x, y, and z values as (0, 0, 0).
- `char enableOffset;`: Declares a character variable `enableOffset` to store the user's choice for enabling the Z-axis offset.
- `std::cout << "Do you want to enable Z-axis offset? (y/n): ";`: Displays a prompt message on the console, asking the user if they want to enable the Z-axis offset.
- `std::cin >> enableOffset;`: Accepts user input from the console and stores it in the `enableOffset` variable.
- `if (enableOffset == 'y' || enableOffset == 'Y') { acc.enableZAxisOffset(); } else { /* Keep Offset Disabled */ }`: Checks if the user's input is 'y' or 'Y'. If it is, the Z-axis offset is enabled using the `acc.enableZAxisOffset()` function of the `Acceleration` class. Otherwise, if the user's input is any other character, the Z-axis offset remains disabled.

This portion of the code interacts with the user, asking for their preference regarding the Z-axis offset. If the user agrees, the Z-axis offset in the `Acceleration` object is enabled, and if not, it remains disabled.

Data Processing Explanation

This segment of code handles the processing of data obtained from the input file:

- `while (std::getline(inputFile, line)) {`: Iterates through each line in the input file `inputFile`.
- `if (line.find("G4") == 0) {`: Checks if the line starts with the identifier "G4".
- `std::string checksum = line.substr(line.length() - 2);`: Extracts the last two characters from the line, assuming these are the checksum.
- The lines 130-135 in code perform a checksum calculation:
  - A loop iterates through the line's characters except for the last two characters.
  - Each character's ASCII value is added to `sum`, and the result is bitwise ANDed with 0xFF to keep the result under 8bit compensating 8bit addition overflow by wrapping it in 8bit.
  - The result is converted to a hexadecimal string `calculatedChecksum`.

  - This calculated checksum is compared with the extracted checksum. If they match, the data is considered valid and further processed.

- If the checksums match:
  - The line is written to the `validChecksumFile` file.
  - X, Y, Z values are extracted from specific positions in the line and parsed as hexadecimal integers.
  - If needed (values are signed), the code adjusts the values using 2's complement.
  - Constants are defined for conversion calculations.
  - Raw values are converted to 'g' units and adjusted if the Z-axis offset is enabled.
  - The `Acceleration` object (`acc`) is updated with these values and used to generate formatted output in the `outputFile`.

- If the checksums don't match:
  - The `invalidChecksumCount` in the `Acceleration` object (`acc`) is incremented.

This part of the code is responsible for parsing each line from the input file, verifying checksums, extracting data, and processing it based on the validity of the checksum.

Processing Valid Data

This segment of the code processes data when the calculated checksum matches the extracted checksum from the line.

- `validChecksumFile << line << std::endl;`: Writes the current line to the `validChecksumFile` file, assuming its checksum is valid.
- `std::string xStr = line.substr(4, 3);`: Extracts a substring of length 3 starting from the 4th character (zero-indexed) in the line and assigns it to `xStr`.
- `int xVal = std::stoi(xStr, nullptr, 16);`: Converts the hexadecimal string `xStr` to an integer `xVal`.
- `if (xVal > 2047) xVal -= 4096;`: Checks and handles 2's complement for signed values.
- `const float maxForce = 32.0f; const float scale_factor = maxForce / 2047.0f;`: Defines constants for conversion calculations.
- `float xG = static_cast<float>(xVal) * scale_factor;`: Converts the raw value `xVal` to 'g' units using the scale factor.
- `if (acc.z_axis_offset_enabled) { zG -= 1.0f; }`: If the Z-axis offset is enabled, adjusts the `zG` value by subtracting 1.0.
- `acc.updateStatistics(xG, yG, zG);`: Updates the statistics in the `Acceleration` object with the converted `xG`, `yG`, and `zG` values.

- `acc.checkAlert(yG);`: Checks for alerts based on the `yG` value.
- `acc.x_g = xG; acc.y_g = yG; acc.z_g = zG;`: Updates the `x_g`, `y_g`, and `z_g` values in the `Acceleration` object.
- `outputFile << acc.getFormattedValues() << std::endl;`: Writes the formatted values of the `Acceleration` object to the `outputFile`.

This part of the code processes and manipulates data when a valid checksum is obtained, converting raw values to 'g' units, handling offsets, and updating statistics in the `Acceleration` object.

Final Data Processing and Output

- `inputFile.close();`, `outputFile.close();`, `validChecksumFile.close();`:
  - Closes the input and output files.
- `std::cout << "Total valid messages: " << acc.valid_messages << std::endl;`:
  - Displays the total count of valid messages processed.
- `std::cout << "Total invalid checksum messages: " << acc.invalid_checksum_messa << std::endl;`:
  - Displays the count of messages that had an invalid checksum.
- `std::cout << "Maximum X-axis value: " << acc.max_x << std::endl;`:
  - Shows the maximum value encountered on the X-axis.
- `std::cout << "Maximum Y-axis value: " << acc.max_y << std::endl;`:
  - Shows the maximum value encountered on the Y-axis.
- `std::cout << "Maximum Z-axis value: " << acc.max_z << std::endl;`:
  - Shows the maximum value encountered on the Z-axis.
- `std::cout << "Total alerts encountered: " << acc.alerts_encountered << std::endl;`:
  - Displays the total number of alerts encountered during processing.
- `return 0;`:
  - Indicates successful execution of the program by returning a status code of 0 to the operating system.

This final segment of the code closes the opened files, prints statistical information about the processed data to the console, and ends the program, indicating successful completion.

Code Flow Diagram