C++ Senior Developer PubSub Test

The purpose of this test is to evaluate your aptitude for implementing C++ solutions to problems similar to those you will likely encounter on a day-to-day basis.

Much of the work we do is very latency-sensitive, and we strive to resolve as much logic as possible at compile time, while keeping flexible interfaces for easy expansion later.

We avoid virtual functions and heap memory allocations unless they are absolutely necessary. This test is a good example of this approach.

Problem Description

- We have 3 key concepts: `FeedPublisher`, `MarketBuilder` and `Strategy`.
- A `FeedPublisher` publishes messages to both `MarketBuilder` and `Strategy`, i.e. `MarketBuilder` and `Strategy` should derive from a base class `FeedSubscriber`, because they share the same interface.
- A `MarketBuilder` receives messages from the `FeedPublisher`, processes the messages into an orderBook, and send that orderBook to a `Strategy`.
- A `Strategy` subscribes to both `FeedPublisher` and `MarketBuilder`, i.e. `Strategy` needs to act on not only different types of messages, but also different types of orderBooks.
- Each electronic exchange requires a different type of `FeedPublisher`, for example `SingaporeExchangeFeedPublisher` and `AmericanExchangeFeedPublisher`.
- Each feed requires a different type of `MarketBuilder`, for example `SingaporeExchangeMarketBuilder` and `AmericanExchangeMarketBuilder`.
- The design allows for easy expansion to other exchanges, such adding `EuropeanExchangeFeedPublisher`, `EuropeanExchangeMarketBuilder`, and other types of messages, such as adding `AmericanExchangeDeleteOrder`, `AmericanExchangeModifyOrder` etc.
- Strategies can listen to multiple exchanges and builders - see `Strategies.h`.

Tasks & formats:

- Fill in the `/* ??? */` in the files to complete the program. Feel free to add/delete classes / files / helper functions etc. to finish the implementation according to design, but be mindful of software bloat & future extendability of the interface.
- Your solution must not introduce virtual table overhead.
- A correct implementation will let the current `Makefile` build & run the program, and the various types of messages and orderbooks are sent, then processed properly by the strategies.

- Your solution must be written in C++. You may use any features of C++ up to C++23. Please use the standard library only; there should be no need for third-party libraries.
- Your solution must build on Linux - please use **Ubuntu 24.10 with GCC 14.2.0**. We suggest making a fresh Virtual machine/docker for this task to fully ensure your solution is reproducible.
- You are allowed to use resources like Google, ChatGPT, or StackOverflow, but you will be asked detailed questions about your implementation if you proceed to the next round. Be sure you deeply understand any code you submit, and take notes of design choices you made along the way to explain your reasoning.
- Optional: to test the extendability of your implementation, you can try to add `EuropeanExchangeFeedPublisher`, `EuropeanExchangeMarketBuilder` and a `ThreeMarketsStrategy` that listens to all events in `SingaporeExchange`, `AmericanExchange` and `EuropeanExchange` and process them. A correct implementation will allow adding addtional exchanges very easily. We will give extra credits if you finish this stage.