CM2005: Object-Oriented Programming

Mid-Term Assignment

Arjun Muralidharan

University of London

10th July 2020

Contents

List of Figures

List of Algorithms

List of Listings

1 General Overview & Approach

The objective of this assigment was to develop an automated trading program that emulates the manual operation of the "Merklerex" starter application. The application in this submission was largely developed independently from the starter code provided, with some parts copied over to ensure correct basic operation. The **key design approach** used in this assignment was to move all core trading logic to a new `MerkleBot` class, and reducing the original `MerkleMain` class to only control the user input and output operations. This new relationship is shown in Figure 1. This meant not only creating new functions to address the requirements of the assignment, but also refactoring much of the existing code to use the new class in order to recude duplication. The bot functionality is invoked as a new, additional menu option of the simulation and can be invoked at the start of the simulation, or after executing one or more manual trades. Once invoked, the rest of the simulation runs in an automated fashion.

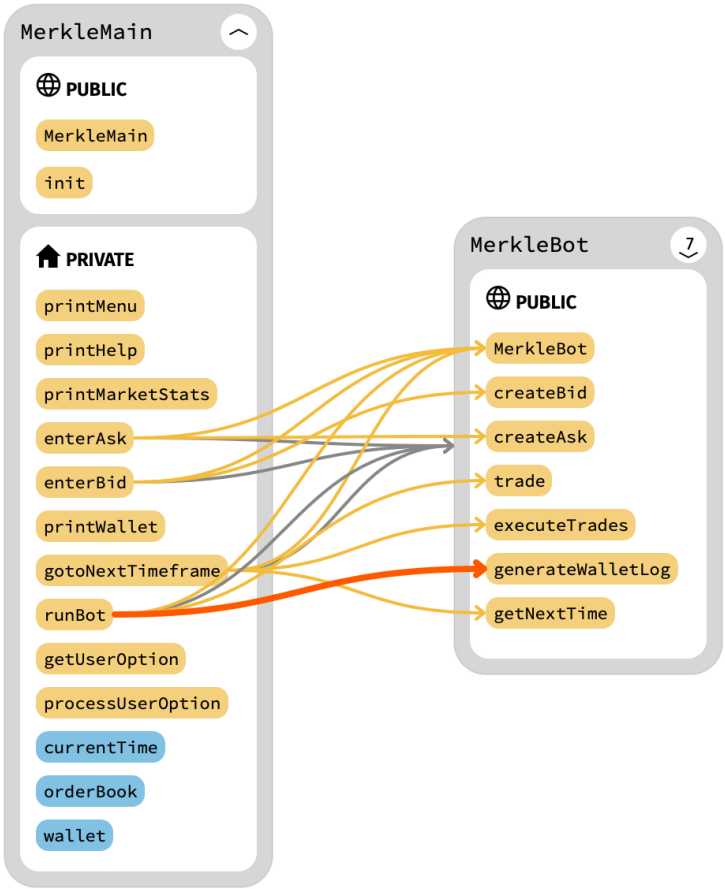

Figure 1. *Relationship between the MerkleMain and MerkleBot classes*

Performance considerations Major refactoring was done to improve performance of the simulation, achieved by using a more efficient data structure to store and operate on the order book data. A `<map>` structure storing timestamps and their respective orders as a vector yielded significant benefits,

as a `<map>` is stored sorted by keys internally, eliminating the need to sort the order book during the simulation and reducing algorithmic time complexity.

More optimisation is possible by pre-loading and storing frequently used subsets of the market data, however for the scope of this assignment the bot operation was left to operate at between 10 and 30 seconds on typical modern-day CPUs.

2 Requirements

This section describes how each of the requirements were addressed with corresponding code extracts where applicable.

2.1 R1: Market Analysis

2.1.1 R1A: Bot accesses order book

The bot operates on the same order book as the main application. This is achieved by passing the order book (as well as the wallet and the current time string) as references to the bot at instantiation. As the bot is instantiated, it initialises variables using an initializer list.

This approach, shown in code in Listing 1 allows the bot to be instantiated only when needed, run any required functions which directly operate on the current, most updated versions of the order book (and the wallet), and then the bot can be destroyed safely.

Listing 1 Accessing the order book by passing by reference

```
class MerkleBot
{
public:
        MerkleBot(Wallet& _wallet, OrderBook& _orderBook, std::string&
        ↪  _currentTime);
        Wallet& wallet;
        OrderBook& orderBook;
        std::string& currentTime;
};
```

```
MerkleBot::MerkleBot(Wallet& _wallet, OrderBook& _orders, std::string&
↪  _currentTime)
        : wallet{_wallet}, orderBook{_orders}, currentTime{_currentTime}
{
}
```

2.1.2 R1B: Bot can generate predictions of future market prices

This requirements was achieved by creating an algorithm that generates recommendations of currencies to buy and sell; a type of "stock picks". Additionally, a price is picked from the current timeframe at which the currency should be bought or sold.

The algorithm, shown in Algorithm 1, is a largely simplified version of using a **simple moving average (SMA)** to judge if the price trend of a given currency is positive or negative. If the trend has been negative for a given period of time, this indicates a sell signal. The opposite is true for a buy signal. This is a conservative trading approach relying on trends over a number of timeframes.

Algorithm 1 Stock Picks Algorithm

```
function StockPicks
    predictions ← new Map(OrderBookType, string);
    buyPredictions ← new Map(string, double);
    sellPredictions ← new Map(string, double);
    for all product ∈ OrderBook.products do
        if Simple Moving Average (20 days) > current high price for asks then
            sellPredictions.push(product, current low price for asks)
        end if
        if Simple Moving Average (20 days) < current high price for bids then
            buyPredictions.push(product, current high price for bids)
        end if
    end for
    predictions.push(bid, buyPredictions)
    predictions.push(ask, sellPredictions)
    return predictions
end function
```

The corresponding code is available in the `StockPicks()` function of the `MerkleBot` class and reflects the necessary C++ syntax to return a valid map containing two submaps (one each for ask and bid predictions) as a result.

2.2 R2: Bids, asks and sales

2.2.1 R2A: Bot can generate bids and place them in the order book

The result of the previous `StockPicks()` function is later called within the `trade()` function. This function loops through the returned stock picks (using a range-based loop on the map with C++17 structured bindings) and creates bids for each stock pick. The specific code extract responsible for this is shown in Listing 2.

Listing 2 Creating asks and bids based on stock picks

```
void MerkleBot::trade()
{
        // ...
        if (bidPicks.size() > 0)
        {
                for (auto const [product, price] : bidPicks)
                {
                        // Get the currency pair for the stock pick product
                        std::string currency = product.substr(product.find("/") +
                        ↪  1);

                        // Get the correct balance from your wallet
                        double balance{wallet.checkBalance(currency)};

                        // Create the bid, splitting your balance across all bids
                        ↪  for the same currency
                        createBid(price, (balance / bidPicks.count(product)) *
                        ↪  price, product);
                }
        }
        // ...
}
```

The bot has a function called `createBid` that creates the actual bid. This function has been slightly refactored from the original starter code so it can be used by both the bot as well as manual operation from the `MerkleMain` class. The code is not reproduced in this document as it is very similar to the starte code and algorithmically unchanged.

2.2.2 R2B: Bot can generate asks and place them in the order book

This is solved identically to requirement *R1A* with the only difference being that the program checks for asks instead of bids.

There is potential to further optimize these procedures so that a single procedure can handle both bids and asks, and the functions `createBid` and `createAsk` could be merged to further reduce code duplication.

2.2.3 R2C: Bot updates wallet after successful sales

The execution of sales and update to the wallet happens as part of the `executeTrades` function. This function has been encapsulated to allow for both manual calls from `MerkleMain` and within the `trade()` function of the bot.

For each `sale` generated by the matching engine (unchanged from the starter code), this function processes the sales using the unchanged `wallet.processSale()` function, which in turn updates the wallet of the application. Because we passed the wallet by reference, it is shared across both the bot and manual modes of the application and only needs to be updated once, as shown in Listing 3.

Listing 3 Updating the wallet as part of executing trades

```
void MerkleBot::executeTrades()
{
//...
        for (OrderBookEntry& sale : sales)
        {
                if (sale.username == "simuser" && wallet.canFulfillOrder(sale))
                {
                        wallet.processSale(sale);
                }
        }
// ...
}
```

2.3 Logging

Logging has been implemented to ouput log data to text files. As part of the logging mechanism, when the application is instantiated, previously existing log files are deleted as part of the `MerkleMain::init()` function. This might be extended to be less destructive, e.g. by dating the file names and making logging an option in the application to activate.

2.3.1 R3A: generates a log of the contents of its wallet

Logging of the wallet has been implemented in a `generateWalletLog` function that simply outputs the wallet contents to the text file, as shown in Listing 4. It is called only once in this application, at the end of the bot's trading routine (`MerkleBot::trade()`). It re-uses the operator overloading implemented in the starter code to output the wallet contents to the output stream.

Listing 4 Logging the wallet contents

```
void MerkleBot::generateWalletLog()
{
        std::ofstream walletLog;
        walletLog.open("walletLog.txt");
        walletLog << wallet << std::endl;
        walletLog.close();
}
```

2.3.2 R3B: Bot generates a log of bids and asks

The logging of bids and asks happens as part of the `MerkleBot::generateBidsAsksSalesLog()` function. This function merges the logging of bids, asks and sales as the treatment is very similar, with the only difference being which file the output is written to, as shown in Listing 5. This function is called whenever the simulation creates a bid, an ask or a sale. It covers both manual and bot modes. If the OrderBookType is not identifiable, the function logs the contents of the order to an errors.txt file. However, this can only occur if the OrderBookType enumeration is expanded and this function does not account for it.

Listing 5 Logging asks and bids

```
void MerkleBot::generateBidsAsksSalesLog(const OrderBookEntry& order)
{
        std::ofstream logFile;
        if (order.orderType == OrderBookType::ask || order.orderType ==
        ↪  OrderBookType::bid)
        {
                logFile.open("bidsAsksLog.txt", std::ios::app);
        }
        else if (order.orderType == OrderBookType::asksale || order.orderType ==
        ↪  OrderBookType::bidsale)
        {
                std::ofstream salesLog;
                logFile.open("sales.txt", std::ios::app);
        }
        else
        {
                logFile.open("errors.txt", std::ios::app);
        }

        logFile << order.timestamp << "," << order.product << "," <<
        ↪  std::setprecision(5) << order.price << "," << std::setprecision(5) <<
        ↪  order.amount << ", "
                        << order.username << std::endl;
        logFile.close();
}
```

3 Code style & technique

The requirements for code style and technique have been fulfilled as follows.

1. **C1** : All code is within the `/merklerex` folder and stored in `.hpp` and `.cpp` files. The only additional files are the compiled application (compiled on an Intel Core i5 CPU), and the market data file as provided in the course. Header files only contain declarations, while implementations are moved to the corresponding implementation files.

2. **C2** : All public functions have comments using the `/** ... */` convention to support modern IDEs. Purpose, inputs and outputs have been described.

3. **C3** : The coding style used is largely based on the Google C++ Style Guide with minor adjustments for tabbing and whitespace. However, it has been applied consistently throughout the application.

4. **C4** : Functions, classes and variables are named as per the Google style guide to describe the purpose or intent of the object. Camel case has been used throughout.

References

[1] Horton, I., & van Weert, P. (2018). *Beginning C++17: From Novice to Professional*. Berkeley, CA: Apress.

[2] Mitchell, C. (2020). *How to Use a Moving Average to Buy Stocks* . Retrieved July 09, 2020, from https://www.investopedia.com/articles/active-trading/052014/how-use-moving-average-buy-stocks.asp