Stock Price Prediction With C++

Vincent Scala

Ozzie Martin

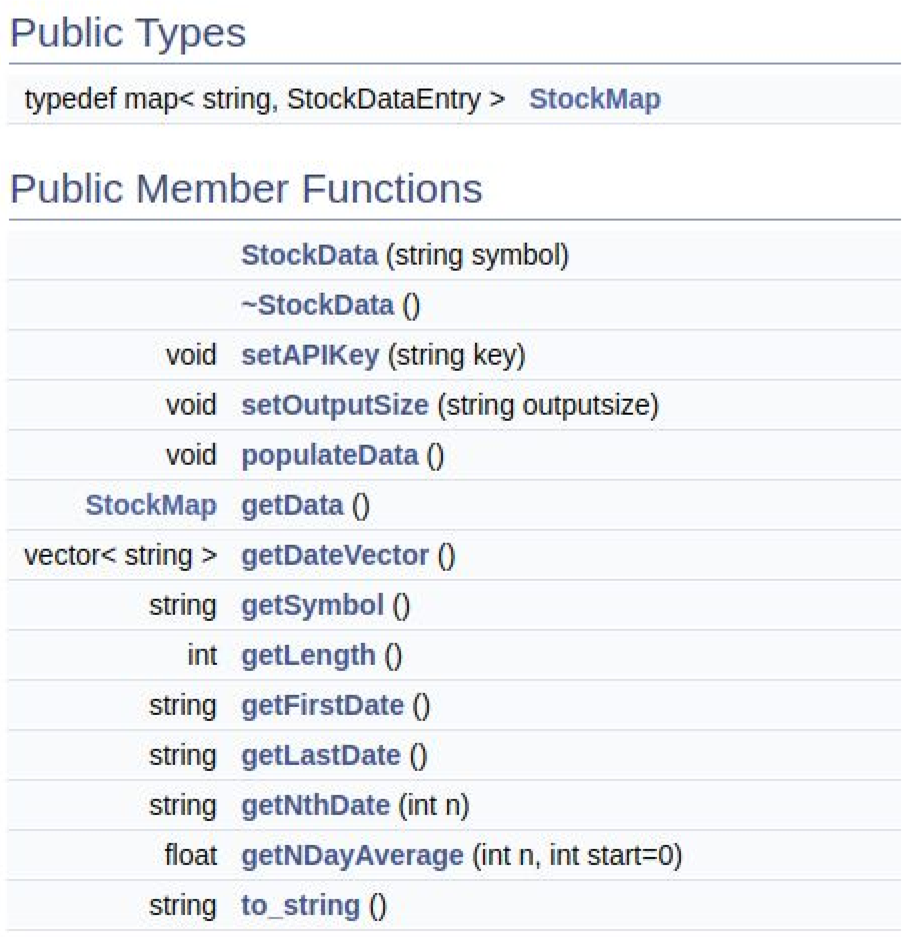
UNC Charlotte

Introduction

Our application set out to predict the future price of a stock on the NYSE. We chose this application because it deals with a real world application in which data is easily attainable, all the while being easily fit into an object oriented model with data structures that can easily interact. The method that we chose to make our predictions, is observing the moving average over a given timespan in days, and making predictions based on those moving averages.

Design

We were able to implement our functionality with one class declaration, ***StockData:***



Public Types

typedef map< string, StockDataEntry >	StockMap

Public Member Functions

	StockData (string symbol)
	~StockData ()
void	setAPIKey (string key)
void	setOutputSize (string outputsize)
void	populateData ()
StockMap	getData ()
vector< string >	getDateVector ()
string	getSymbol ()
int	getLength ()
string	getFirstDate ()
string	getLastDate ()
string	getNthDate (int n)
float	getNDayAverage (int n, int start=0)
string	to_string ()

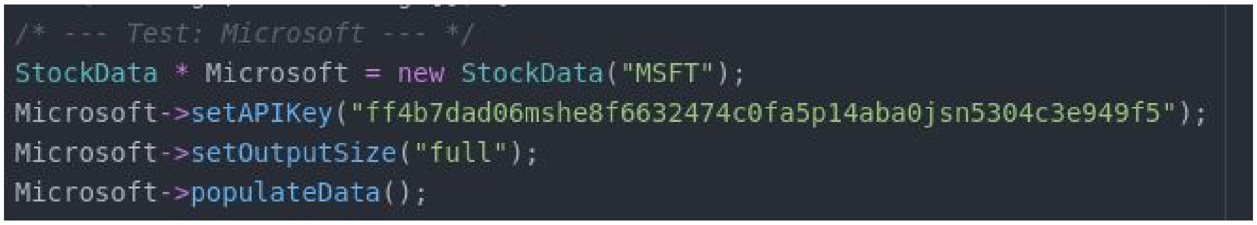
StockData does lots useful things:

- Specifies the symbol of the intended stock to be calculated, and observed by passing it's symbol to the constructor
- Specifies the rapidapi key to be used for data collection with the function ***setAPIKey***
- Specifies the type ***StockMap*** which is a vector of structs defined inside of StockData called ***StockDataEntry*** which is an object representation of a day on the stock market holding data such as
    - Open
    - Close
    - High
    - Low

-
    - Date
    - Volume
    - Adjusted_close
    - Dividend
    - Split_coefficient
- Specifies the ***populateData()*** method which uses curl to communicate with the rapidapi to get JSON data about the requested amount of time on the market
    - At this point in the project, it was decided that if we were to save the data in some type of file for the code to iterate over, that it was going to be a CSV. We went with this choice because working with rapidjson would add un-necessary complexity to the application.

User Interface/Features

Once a user has decided which symbol they wish to predict, they can then pass it to the constructor for the StockData class, and then set the other required parameters as such:

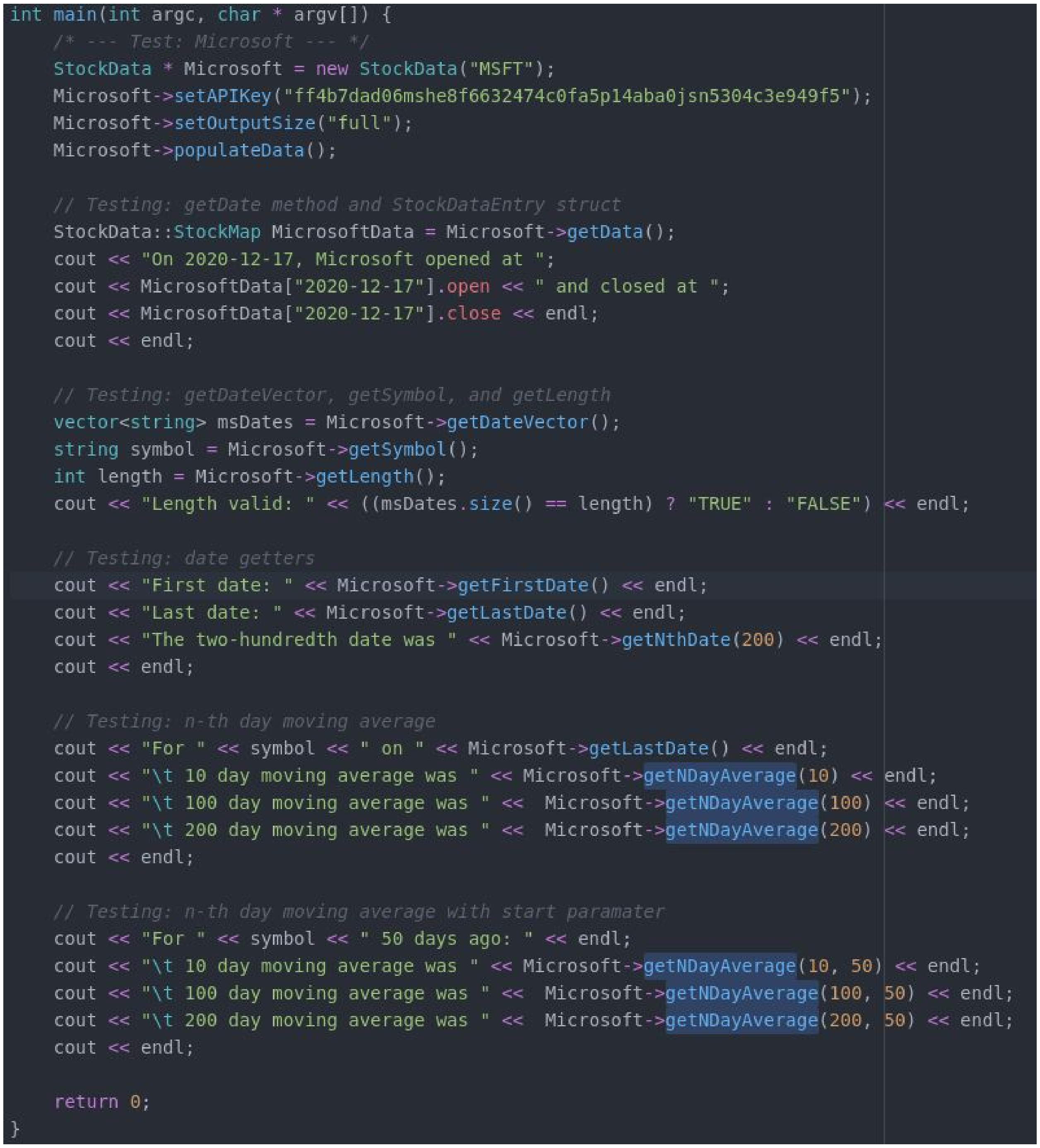


```
/* --- Test: Microsoft --- */
StockData * Microsoft = new StockData("MSFT");
Microsoft->setAPIKey("ff4b7dad06mshe8f6632474c0fa5p14aba0jsn5304c3e949f5");
Microsoft->setOutputSize("full");
Microsoft->populateData();
```

Once these parameters have been passed to the StockData class, we can begin using the functions within the class with which we can put into a test harness to examine further

Testing

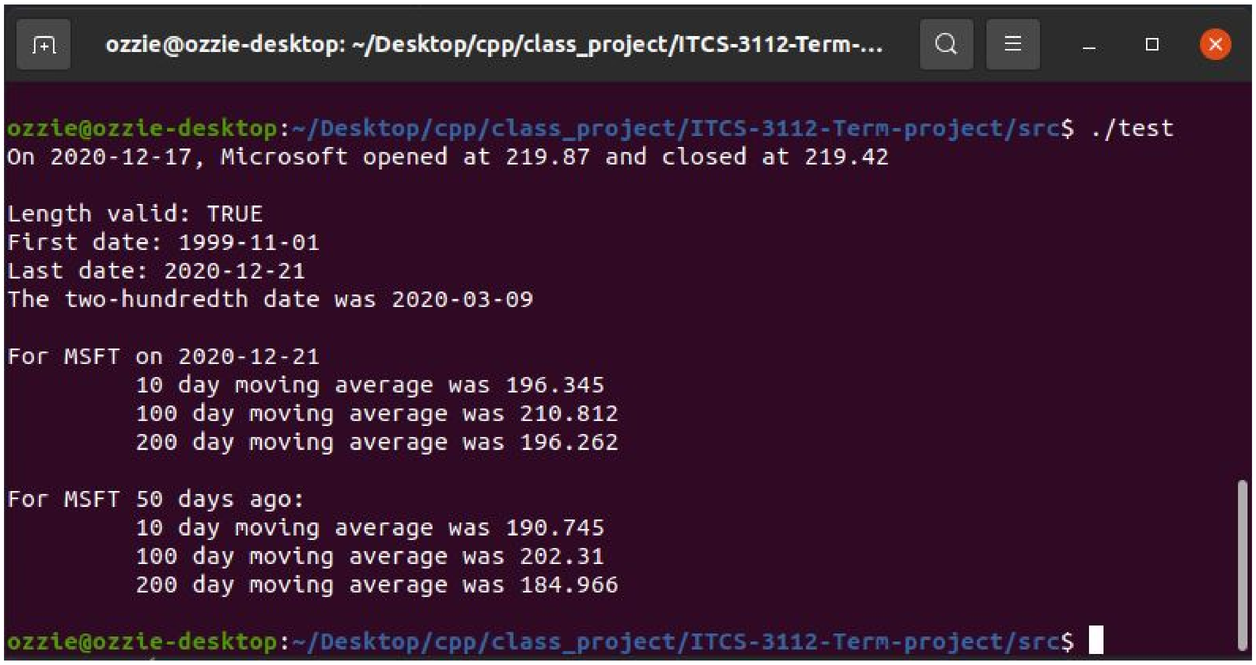
Our test harness is as follows:



```cpp
int main(int argc, char * argv[]) {
    /* --- Test: Microsoft --- */
    StockData * Microsoft = new StockData("MSFT");
    Microsoft->setAPIKey("ff4b7dad06mshe8f6632474c0fa5p14aba0jsn5304c3e949f5");
    Microsoft->setOutputSize("full");
    Microsoft->populateData();

    // Testing: getDate method and StockDataEntry struct
    StockData::StockMap MicrosoftData = Microsoft->getData();
    cout << "On 2020-12-17, Microsoft opened at ";
    cout << MicrosoftData["2020-12-17"].open << " and closed at ";
    cout << MicrosoftData["2020-12-17"].close << endl;
    cout << endl;

    // Testing: getDateVector, getSymbol, and getLength
    vector<string> msDates = Microsoft->getDateVector();
    string symbol = Microsoft->getSymbol();
    int length = Microsoft->getLength();
    cout << "Length valid: " << ((msDates.size() == length) ? "TRUE" : "FALSE") << endl;

    // Testing: date getters
    cout << "First date: " << Microsoft->getFirstDate() << endl;
    cout << "Last date: " << Microsoft->getLastDate() << endl;
    cout << "The two-hundredth date was " << Microsoft->getNthDate(200) << endl;
    cout << endl;

    // Testing: n-th day moving average
    cout << "For " << symbol << " on " << Microsoft->getLastDate() << endl;
    cout << "\t 10 day moving average was " << Microsoft->getNDayAverage(10) << endl;
    cout << "\t 100 day moving average was " <<  Microsoft->getNDayAverage(100) << endl;
    cout << "\t 200 day moving average was " <<  Microsoft->getNDayAverage(200) << endl;
    cout << endl;

    // Testing: n-th day moving average with start paramater
    cout << "For " << symbol << " 50 days ago: " << endl;
    cout << "\t 10 day moving average was " << Microsoft->getNDayAverage(10, 50) << endl;
    cout << "\t 100 day moving average was " <<  Microsoft->getNDayAverage(100, 50) << endl;
    cout << "\t 200 day moving average was " <<  Microsoft->getNDayAverage(200, 50) << endl;
    cout << endl;

    return 0;
}
```

Testing cont.

Here in this screengrab of our test harness, we can see the user specifies the StockData constructor to use the symbol "MSFT" to indicate the microsoft stock price, so the pointer is named Microsoft, and on the object we can see that various methods throughout the StockData class are being called, such as getNDayAverage(int), getNDayAverage(int, int), as well as getLastDate, and getFirstDate. Testing the class with these inputs yielded these results:


ozzie@ozzie-desktop: ~/Desktop/cpp/class_project/ITCS-3112-Term-...

```
ozzie@ozzie-desktop:~/Desktop/cpp/class_project/ITCS-3112-Term-project/src$ ./test
On 2020-12-17, Microsoft opened at 219.87 and closed at 219.42

Length valid: TRUE
First date: 1999-11-01
Last date: 2020-12-21
The two-hundredth date was 2020-03-09

For MSFT on 2020-12-21
         10 day moving average was 196.345
         100 day moving average was 210.812
         200 day moving average was 196.262

For MSFT 50 days ago:
         10 day moving average was 190.745
         100 day moving average was 202.31
         200 day moving average was 184.966

ozzie@ozzie-desktop:~/Desktop/cpp/class_project/ITCS-3112-Term-project/src$
```

Issues/Status

The only major bug right now are memory problems when creating multiple StockData objects, other than this all functions run correctly. Some methods could also be better written in terms of both clarity and speed.

Conclusion

Overall, I think we have a really solid implementation of a moving average calculator for any given day for any given symbol, this could be expanded on to calculate the moving average over a more meticulous spectrum of dates to form an application that could have price prediction capabilities.