C++ FOR FINANCE

ASSIGNMENT 2: FINANCIAL PLANNER

Student ID: 2031611

MSc Mathematical Finance

APRIL 27, 2020

UNIVERSITY OF BIRMINGHAM

Lecturer: Laurence A. Hurst

Introduction

The objective of this assignment is to write a program in C++ 14 to produce a personal financial planner. Only C++ 14 standard libraries were used to write the program. The program will create financial plans for a group of clients, whose aim is to maximise the profit on some low risk product investments before their retirement. Client data and product feed will be provided as input files. Each client's financial plan will be store as a CSV file.

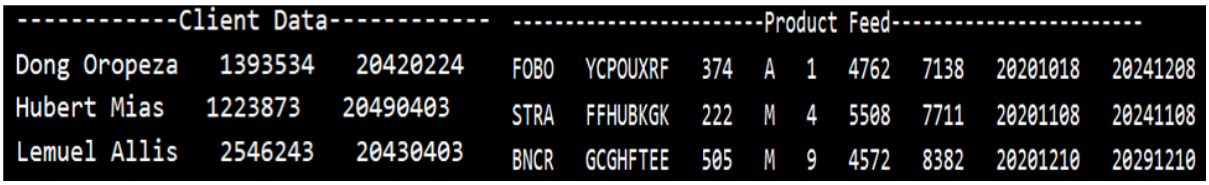

Figure 1 shows the example client data and product feed.

The program is hypothetically going to run continuously until the last client retires, taking a continuous feed of product data to manage their capital to maximise the returns. It can only read the product feed sequentially while making investment decisions. This is to simulate how the product would be available in the real world.

The client data contains:

1. Name (string - may contain spaces).
2. Current capital in pence (int) (i.e. cash immediately available).
3. Expected retirement date (string in the form YYYYMMDD - e.g. 20201225 is 25th December 2020)

The product feed contains:

1. Institution code (4 alphabetical characters)
2. Product code (8 alphabetical characters)
3. Product AER interest rate * 100 (5 digits - for example 00522 is 5.22%)
4. Annual or Monthly interest (1 character - A or M respectively)
5. Product tie-in period in years (2 digits)
6. Product minimum investment in pounds (6 digits)
7. Product maximum investment in pounds (6 digits)
8. Product available from (8 digits - YYYYMMDD)
9. Product maturity date (8 digits - YYYYMMDD)

Approach to the Problem

The goal is to maximise the returns before each client retires. Considering the products are zero risk and the interest rate's compounding effect, investing all of the capital on the highest interest rate product among the available products is considered to be the best. If there is capital remaining after buying the highest rate product, invest the remaining capital on the next highest rate product and so on.

The idea of this strategy is to minimise the amount of capital left uninvested. The initial plan was to invest a portion of capital and save some capital for future, potentially higher rate product. However, it was discovered that this would yield less profit since the higher rate product's advantage would be undermined by the non-profit-generating holding period of the uninvested capital.

Apart from having highest interest rate, the product must also have maturity or tie-in date before client's retirement date. Otherwise the product cannot be redeemed or sold before retirement and hence results in loss of capital and interest accrued on that product.

To further optimise the returns, selling product in the portfolio with interest rate lower than available, uninvested products to free up some capital when the capital is low was considered initially. However, this idea was abandoned eventually because it generated a lower profit than before. There are some modifications could be done to improve this strategy but due to time constraint it was not possible to include it in the program. For example, instead of selling 100% of the lowest rate product, the program should just sell a portion of the amount. Furthermore, the program should determine how far the product is from its next completed anniversary. This way, accrued interest during that uncompleted period would not be lost. (The selling part of the program was not deleted and commented in line 97-157 of Investment.cpp)

Algorithm Summary

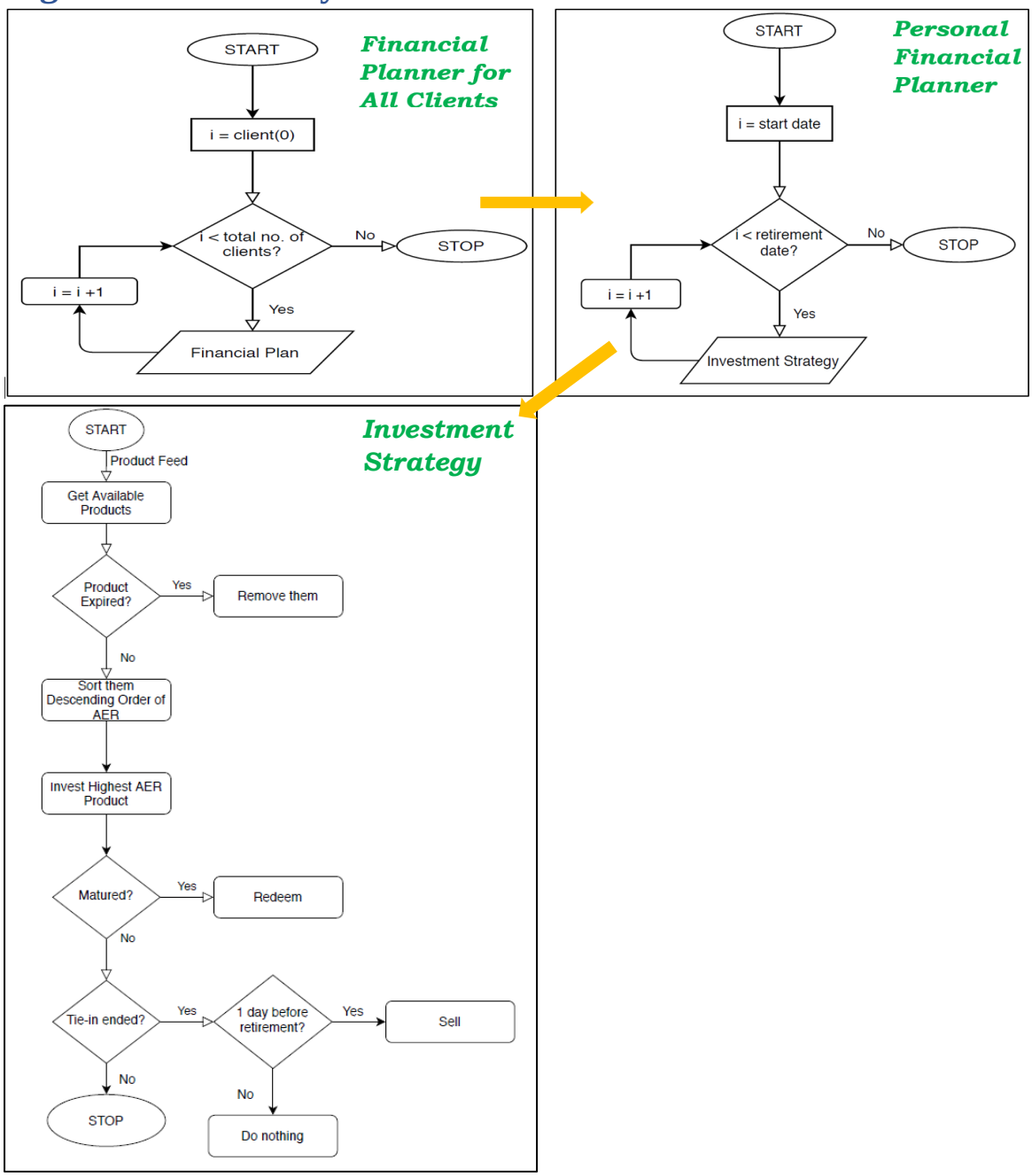

Figure 2 shows the flow chart of financial planner program with orange arrows showing the for loop's hierarchical relationships.

Design-level Efficiency

This algorithm is fairly efficient as the critieria of the product selection is relatively little (select max rate product subject to maturity and tie-in period occurs before retirement). However, there are some changes could be made to the way that it loops through the clients and product feed to improve its efficiency. For example, instead of looping from start date to retirement date and repeat this for all the clients, the same algorithm can be implemented by including investing for all clients as it loops from start date until the last client retires.

Since the main objective of this project is to maximise returns. Efficiency can be compromised a bit more to extract more profits from the strategy. For example, further develop the selling strategy to free up capital in favour of better investments.

All the tests were run on machine with i5-7200U (up to 3.18Hz), 8GB Memory.

Language-level Efficiency

In terms of language-level efficiency, passing arguments by reference into functions were used instead of passing arguments by value wherever possible. This way no copies of arguments were created. Tests were run to compare the efficiencies. It was observed that passing by reference improves the program speed by at least 7% (Pass by reference took 280s ; Pass by value took 300s).

The other programming practice to improve efficiency is choosing pre-fix increment/decrement over post-fix increment/decrement in "*for loop*". In the case of pre-fix, the object just increase/decrease itself without creating a copy.

"*std::vector*" was chosen to be the data container as elements can be accessed at random position in constant time. This is a particularly important for product selection as the best product may not come chronologically. On the other hand, using "*emplace_back*" instead of "*push_back*" also helps with the program's efficiency as the former is always at least as efficient as the latter.

For safety, all the variables or function input arguments that will not be changed during program execution are marked "const" to prevent unintended overwriting. This is a good programming practice to follow and is especially important for financial applications. When accessing an element in vector, "*.at*" was used as it does bound checking and hence it is safer.

Possible Future Improvements

There are some possible improvements on the program given enough time and C++ experience. Developing the program in full object-oriented programming way is useful for maintainability and features addition should the problem becomes more realistic. Moreover, optimising the selling strategy to free up capital for better investments could also be done. As for language-level efficiency, smart pointers should also be used in places where possible.

References

[1] Hurst, L. A. (2020). "C++ For Finance – Assignment 2", University of Birmingham.