ECE 150 - Project - Problem Description - Mo Money

Project submission due: 11:59pm **Friday December 2nd**

Submission filename: History_Transaction_definitions.cpp

1. Introduction

Your first co-op happens to be at an investment firm called Mo-Money. Mo-Money is a newcomer in Canada, and it is primarily interested in deploying a new service to Canadians known as a roboadvisor. Robo-advisors are web-based platforms that automate entire financial planning and investment services from asset allocation, purchasing investment products, rebalancing, and tax reporting with virtually no human involvement. Mo-Money only deals with one specific type of investment product known as exchange-traded funds (ETFs). These are low-cost investment products that are often identified by their ticker symbols on the stock exchange. For example, the Vanguard Growth Portfolio ETF is found by VGRO, and BMO Aggregate Bond Index ETF is found by ZAG on the stock exchange. You can search for it on https://finance.google.ca.

One important part of the service that the robo-advisor provides is generating documents for tax reporting to the Canadian Revenue Agency (CRA). You will implement one particular component of this service known as adjusted cost base (ACB) reporting for ETFs, which is essential when filing taxes that report capital gains or losses on ETF investments. The reason for its importance is that incorrectly computing ACB may result in the client either overpaying in taxes or underpaying; both of which are undesirable for the client.

2. Tasks

In this project, your task is to complete the implementation of two classes: History and Transaction. You will do this by writing the definitions of a number of member functions for these classes. You are provided with the class declarations as well as some of the more basic definitions for these classes. In addition, you are provided with a set of helper functions that enable you to read transaction data from a file.

The following list outlines the high-level tasks in the recommended order for completion.

- Setup project in IDE
- Add skeleton functions
- Ensure that code compiles successfully
- Implement Transaction and History constructors
- Implement overloaded less-than operator
- Implement the `History::insert(...)` function to add a node to the linked-list
- Implement the `History::read_history()` function to create the linked-list using the input file
- Code de-allocation of the linked-list so that there aren't any memory leaks
- Implement the `History::print()` function so that we can visually verify the state of the linked-list after its creation.
- Implement the `History::sort_by_date()` function so that the linked-list is re-ordered chronologically
- Implement the `History::update_acb_cgl()` function so that key financial metrics are calculated
- Implement the `History::compute_cgl()` function so that key financial metrics are calculated

3. The Transaction class

The `Transaction` class will store information pertinent to a single transaction. Each Transaction object is a node of the linked-list.

Private member variables

The `Transaction` class contains the following private member variables.

```
unsigned int trans_id;
```

Unique transaction identifier for every transaction.

```
unsigned int day;
```

Day of the trade date.

```
unsigned int month;
```

Month of the trade date.

```
unsigned int year;
```

Year of the trade date.

```
std::string symbol;
```

Ticker symbol.

```
std::string trans_type;
```

Transaction type. Value must be either "Buy" or "Sell".

```
unsigned int shares;
```

Number of shares involved in the transaction.

```
double amount;
```

Total dollar amount of the transaction. Always positive.

```
Transaction *p_next;
```

Stores the address of the next `Transaction` in the linked-list.

```
static unsigned int assigned_trans_id;
```

A static class variable used to generate unique identifiers whenever a transaction object is created. This is the identifier you will use to set the `trans_id` private member, and increment it after the assignment. A static class variable is a single variable that is shared amongst all instances of the class.

For local testing purposes, initialization of `assigned_trans_id` is included in the starter main.cpp file. For testing on Marmoset, the tests will initialize this variable at the start of each test.

Public member functions

The `Transaction` class contains the following public member functions.

```
Transaction( std::string ticker_symbol, unsigned int day_date, unsigned int month_date,
unsigned year_date, bool buy_sell_trans, unsigned int number_shares, double trans_amount
);
```

This constructor assigns the parameters to their respective `private` member variables and appropriately initializes all other private member variables. For finding the appropriate value for the transaction id, refer to the `assigned_trans_id` member variable. To be defined by the student.

```
~Transaction();
```

Destructor. To be defined by the student.

```
bool operator<( Transaction const &other );
```

This overloaded operator implements the less-than operator for comparing Transactions, based on which Transaction occurred first chronologically. Returns true if and only if the transaction date for the left operand occurs prior to the transaction date of the right operand. If the transaction dates are equal for both left and right operands, returns true if and only if the transaction id of the left operand is *~~less~~ greater than the transaction id of the right operand.

***EDIT: 11-09-2022** -- There is a mistake in the Marmoset testing code, which compares the transaction ids in reverse (i.e. greater than rather than less than). We are changing the specification rather than forcing students who have submitted to re-submit.

This function is useful to determine sort order when sorting Transactions. To be defined by the student.

```
std::string get_symbol() const;
```

Returns the ticker symbol. Already defined.

```
unsigned int get_day() const;
```

Returns the day of the date the transaction occurred. Already defined.

```
unsigned int get_month() const;
```

Returns the month of the date the transaction occurred. Already defined.

```
unsigned int get_year() const;
```

Returns the year of the date the transaction occurred. Already defined.

```
unsigned int get_shares() const;
```

Returns the number of shares for the transaction. Already defined.

```
double get_amount() const;
```

Returns the amount paid or earned for the transaction. Already defined.

```
double get_acb() const;
```

Returns the adjusted cost base for the transaction. Already defined.

```
double get_acb_per_share() const;
```

Returns the adjusted cost base per share. Already defined.

```
unsigned int get_share_balance() const;
```

Returns the share balance. Already defined.

```
double get_cgl() const;
```

Returns the capital gain or loss for that transaction. Already defined.

```
bool get_trans_type() const;
```

Returns `true` if the transaction is a Buy and `false` if it is a Sell. Already defined.

```
unsigned int get_trans_id() const;
```

Returns the transaction identifier. Already defined.

```
Transaction *get_next();
```

Returns the `p_next` pointer. Already defined.

```
void set_acb( double acb_value );
```

Sets the ACB private member variable. Already defined.

```
void set_acb_per_share( double acb_per_share_value );
```

Sets the ACB per share private member variable. Already defined.

```
void set_share_balance( unsigned int balance );
```

Sets the share balance. Already defined.

```
void set_cgl ( double value );
```

Sets the capital gain or loss private member variable with the parameter value. Already defined.

```
void set_next( Transaction *p_new_next );
```

Sets the `p_next` pointer to `p_new_next` in the linked list. Already defined.

```
void print();
```

Prints the transaction information to the console output. This provided function helps ensure that console output meets a standard format for printing Transaction details. Already defined.

4. The History class

The `History` class contains the head pointer that holds the complete linked-list. It also contains functions for creating and manipulating the linked-list.

Private member variables

The `History` class will have the following private member variables.

```
Transaction *p_head;
```

A pointer to a transaction that denotes the beginning of the linked list.

Public member functions

The `History` class will have the following public member functions.

```
History();
```

Default constructor. Initializes linked-list to be empty. To be defined by the student.

```
~History();
```

Destructor. To be defined by the student.

```
void insert( Transaction *p_new_trans );
```

Adds the passed `Transaction` instance to the end of the linked-list of Transactions. To be defined by the student.

```
void read_history();
```

Reads the transaction history from the input file (`transaction_history.txt`) and creates the linked-list. Helper functions for reading the input file are provided in the project4.hpp/project4.cpp files. See section on *Helper Functions - File Reading* for details on provided helper functions.

This function calls the `insert(...)` function to add the new Transaction to the linked-list. The Transactions are to be added to the linked-list in the same order that they are read from the input file.

The transactions in the input file are not necessarily in any sorted order. Refer to the provided sample `transaction_history.txt` for an example of an input file.

To be defined by the student.

```
void print();
```

Prints to the console output the transaction history. To be defined by the student.

```
void sort_by_date();
```

Sorts the linked list in ascending order of trade date (or transaction id). Uses the overloaded less-than operator to define sort order. To be defined by the student.

```
void update_acb_cgl();
```

Walks through the linked-list and updates the following private member variables: `acb`, `acb_per_share`, `share_balance`, and `cgl` for each Transaction. See section on *Computing gains and losses using Adjusted Cost Base* for additional details on how these values are calculated. To be defined by the student.

`double compute_cgl( unsigned int year );`

Computes the capital gains or capital losses for every transaction in the history of transactions and updates the respective instances in the linked list. In addition, this function returns the total capital gains for the specified year. See section on *Computing gains and losses using Adjusted Cost Base* for additional details on how to the CGL is calculated. To be defined by the student.

5. Project File Architecture

This project contains a number of files. In general, HPP files (called "header files") contain class and function declarations, whereas CPP files contain function definitions. All of these files should be in the project folder.

`main.cpp`

Contains the main() function. Used for local testing. Student is encouraged to modify this file for local testing purposes.

`History.hpp`

History class declaration. Should not be modified.

`Transaction.hpp`

Transaction class declaration. Should not be modified.

`project4.hpp`

Declaration of helper functions for file reading. Should not be modified.

`project4.cpp`

Definition of helper functions for file reading. Should not be modified.

`transaction_history.txt`

Input text file containing transaction data. Student is encouraged to modify this file for local testing purposes.

`History_Transaction_definitions.cpp`

Definitions for both History and Transaction member functions. Note that in a professional programming codebase these would be separated into two files. However, in order to simplify the process for submitting to Marmoset, this has been combined into a single file. Student is required to modify this file.

Visual Studio Code Project Setup

1. Put **"Transaction.hpp"**, **"History.hpp"**, **"History_Transaction_definitions.cpp"**, **"project4.hpp"**, **"project4.cpp"**, **"main.cpp"**, **"transaction_history.txt"** in a **new** folder you create.
2. Start by creating the skeleton definitions of every functions (methods) in your **History_Transaction_definitions.cpp** file as stated in Your Tasks.
3. To test your **History_Transaction_definitions.cpp** file, type the following command into your vscode integrated terminal while in the folder directory.

```
g++ -o main History_Transaction_definitions.cpp project4.cpp main.cpp -std=c++11
```

5. Run main to run the test file. (`./main`)

If you want to use the old method of compiling your code with `Run Build Task` shown in week 1 , you can still do that by going to `Terminal > Configure Tasks..` and switching your args entry to the following:

```
"args": [
    "-std=c++11",
    "-fdiagnostics-color=always",
    "-g",
    "${file}",
    "project4.cpp",
    "main.cpp",
    "-o",
    "${fileDirname}\\main.exe"
],
```

6. Helper Functions - File Reading

You can use the following functions to read the input from a text file. These functions are provided in a file called `project4.hpp`. Notice that there is an `ece150::` similar to the standard library namespace `std::` that is used for calling each of these functions.

Accessing a transaction from the input file.

`void ece150::open_file()`

This opens the file named `transaction_history.txt` for reading. This function must be called before any of the provided functions.

`void ece150::close_file()`

This closes the file named `transaction_history.txt` for reading. This function must be called once the reading of the file is complete.

`bool ece150::next_trans_entry()`

This function allows the reading of the next transaction in the input file. This function returns `false` if there are no more transactions to be read, and `true` otherwise. It is an undefined operation to call this function if `ece150::open_file()` has not yet been called (meaning, the behavior is undefined; it may return garbage, it may throw an error, etc.).

This function steps through the input file one transaction at a time. There is no way to go back to the previous transaction once the `next_trans_entry()` is called.

Accessing individual inputs of a given transaction.

All the functions below are reading different fields of the transaction entry. Therefore, you must have requested the reading of a valid transaction using `next_trans_entry()` prior to calling the functions below.

`std::string ece150::get_trans_symbol()`

This function returns the ETF symbol as a `std::string` in the transaction entry.

`unsigned int ece150::get_trans_day()`

This function returns the day of the trade date for the transaction being read from the file.

`unsigned int ece150::get_trans_month()`

This function returns the month of the trade date for the transaction being read from the file.

`unsigned int ece150::get_trans_year()`

This function returns the year of the trade date for the transaction being read from the file.

`unsigned int ece150::get_trans_shares()`

This function returns the number of shares bought or sold for the transaction being read from the file.

`bool ece150::get_trans_type()`

This function returns `true` if the transaction type is a Buy and `false` if it is a Sell for the transaction being read from the file.

`double ece150::get_trans_amount()`

This function returns the amount paid or the amount received for the transaction being read from the file.

7. Sorting the linked-list

You should use the following approach to sort the linked list. Let us call the *original* linked list as the one that you populated in the order of input from the file, and *sorted* linked list as the resulting one from this member function. You will sort by simply re-organizing the transactions; thus, no additional dynamic allocation of `Transaction` instances will be done. You can use the following approach:

1. Remove the first transaction from the original linked list, but do not delete it.
2. Reconnect the original linked list so that what was previously the second transaction is now the first transaction, and the number of transactions in the original linked list is reduced by one.
3. Insert the removed transaction into the sorted linked list such that it is inserted in its correct position with respect to the trade date.
4. Repeat these steps for the remaining transactions in the original linked list.

8. Easy as ACB: Computing gains and losses using Adjusted Cost Base

We will use an example to illustrate the steps necessary to compute the ACB. Suppose that MoMoney has a client called Rupert who has purchased a different number of shares of VGRO over the course of the year. A share is simply a unit of the product. For example, on January 10, 2018, Rupert bought 150 shares of VGRO for the total amount of $10300.14. Throughout the year, Rupert made additional purchases of the same ETF. Below is Rupert's purchase history for the year for the VGRO ETF.

Trade date	Transaction Type	Shares	Amount Paid
10-Jan-18	Buy	150	$10,300.140
24-Feb-18	Buy	85	$7,423.050
8-Aug-18	Buy	43	$3,367.760
11-Nov-18	Buy	78	$7,028.580

In December of 2018, Rupert needed to sell some of his shares. He made two sells that are shown below with the amounts he received for each sell transaction.

Trade date	Transaction Type	Shares	Amount Paid
8-Dec-18	Sell	55	$5,958.150
22-Dec-18	Sell	80	$2,817.600

For tax reporting purposes, whenever a share is sold, the CRA needs the client to report if there are any capital gains or losses (this is a simplification, please consult the appropriate tax documentation for details – not required to complete this project). A capital gain is an increase in value of the investment product, and a capital loss is a loss in the value of the investment product. These are important because the client must pay a tax on the capital gains to the CRA, and on a loss, the client can claim it as a deduction.

We are only interested in computing the capital gains or losses incurred due to any shares being sold. In order to make this computation, we need to maintain the ACB, and its ACB per share for every buy and sell transaction. Below are the computed values of the ACB, share balance after the transaction, and the ACB/Share for every Buy performed for Rupert.

Note: it is important that the transactions in a year must be sorted in ascending order of the trade date in order to correctly perform the computations.

Purchasing shares

Trade date	Transaction Type	Shares	Amount Paid	ACB	Share Balance	ACB/Share
10-Jan-18	Buy	150	$10,300.140	$10,300.140	150	$68.668
24-Feb-18	Buy	85	$7,423.050	$17,723.190	235	$75.418
8-Aug-18	Buy	43	$3,367.760	$21,090.950	278	$75.867
11-Nov-18	Buy	78	$7,028.580	$28,119.530	356	$78.987

To compute the **ACB** on a Buy, we simply accumulate the total amount paid for the purchases. For example, Rupert's purchase on 24 February 2018 of 85 shares required him to pay $7423.05. The ACB after this purchase is the sum of his first purchase cost of $10300.14 added with $7423.04, which amounts to $17723.19.

To compute the **Share Balance** on a Buy, we add the share balance from the previous transaction with the number of shares purchased with the new transaction. For example, Rupert's purchase on 24 February 2018

resulted in 85 new shares being purchased. Thus, the previous share balance of 150 is added to 85 to give a share balance of 235.

To compute the **ACB/Share** on a Buy, divide the ACB by the Share Balance. For example, the ACB/Share for Rupert's 24 February 2018 transaction was computed as follows: (17723.190/235), which gives $75.418.

Notice the importance of the ACB/Share: since purchases of an ETF may take place multiple times throughout the year with different prices being paid for them, the ACB/Share reflects the average cost for each share over the history of the stock.

Selling shares

For transactions that Sell shares of the ETF, we need to update the ACB, Share Balance, and ACB/Share. In addition, because the client receives investment income from the sell, we have to compute the Capital Gain and Loss (CGL). The table below shows the sells appearing after all the purchases we already saw earlier.

Trade date	Transaction	Shares	Amount Paid	ACB	Share Balance	ACB/Share	Capital Gains/Loss
10-Jan-18	Buy	150	$10,300.140	$10,300.140	150	$68.668	$0.000
24-Feb-18	Buy	85	$7,423.050	$17,723.190	235	$75.418	$0.000
8-Aug-18	Buy	43	$3,367.760	$21,090.950	278	$75.867	$0.000
11-Nov-18	Buy	78	$7,028.580	$28,119.530	356	$78.987	$0.000
8-Dec-18	Sell	55	$5,958.150	$23,775.22	301	$78.987	$1,613.84
22-Dec-18	Sell	80	$2,817.600	$17,456.23	221	$78.987	-$3,501.40

To compute the **ACB** on a Sell, subtract the number of shares sold multiplied (55) by the **ACB/Share** resulting from the previous transaction ($78.987 from the Buy on 11 November, 2018). For example, the Sell transaction on 8 December 2018, the following computation is done: $28119.53 – (55*78.987), which is equal to $23775.22.

The **Share Balance** is simply the previous Share Balance minus the number of shares sold. For the Sell transaction on 8 December 2018, 356 – 55 gives 301 as the new Share Balance.

Notice that the **ACB/Share** for the Sell is computed the same way as the Buy transaction. That is, $23775.22/301, which results in $78.987. Note that the ACB/Share only changes with purchases, and not on sells.

On a Sell, we must compute the Capital Gain and Loss **(CGL)**. To compute the CGL, we subtract the number of shares sold multiplied by the ACB/Share from the previous transaction. For the transaction on 8 December 2018, we perform the following subtraction: $5958.15 – (55*$78.987), which results in a capital gain of

$1613.84. A positive CGL value means that the Sell resulted in capital gains, and a negative CGL value means that the Sell resulted in a capital loss.

Note the transaction on 22 December 2018. Rupert's sell of 80 shares at an unfavourably low price resulted in him receiving $2817.60 for the transaction. This was because of an unfortunate stock market downturn. Once again, using the approach above, the resulting CGL is $2817.60 – (80*$78.987), which gave -$3501.40.; a capital loss.

Since taxation reporting is done on a yearly basis, the CRA needs to know if there was a total capital gain or capital loss in that year. This is typically computed by summing up the CGL values for the entire calendar year. For this example, since there was one Sell transaction that resulted in a capital gain, and another Sell in a capital loss, the total CGL to be reported is -$1877.55.

9. Sample Output

For the following entries in the `transaction_history.txt`.

```
VGRO    10  01  2018    Buy 150 10300.140
VGRO    24  02  2018    Buy 85  7423.050
VGRO    08  08  2018    Buy 43  3367.760
VGRO    11  11  2018    Buy 78  7028.580
VGRO    08  12  2018    Sell    55  5958.150
VGRO    22  12  2018    Sell    80  2817.600
VGRO    04  01  2019    Buy 65  3257.150
VGRO    07  05  2019    Buy 65  4557.150
VGRO    14  06  2019    Sell    80  4451.200
VGRO    16  07  2019    Buy 25  1752.750
VGRO    19  07  2019    Sell    90  6780.600
VGRO    20  10  2019    Buy 100 9011.000
```

You can use the following file with the `main()` to run your implementation.

```
#include <iostream>
#include "project4.hpp"
#include "History_Transaction_definitions.cpp"

#ifndef MARMOSET_TESTING
unsigned int Transaction::assigned_trans_id = 0;
int main() {
    History trans_history{};
    trans_history.read_history();

    std::cout << "[Starting history]:" << std::endl;
    trans_history.print();
    trans_history.sort_by_date();

    std::cout << "[Sorted          ]:" << std::endl;
    trans_history.print();
```

```
    trans_history.update_acb_cgl();
    trans_history.print();

    std::cout << "[CGL for 2018    ]: " << trans_history.compute_cgl(2018) <<
std::endl;
    std::cout << "[CGL for 2019    ]: " << trans_history.compute_cgl(2019) <<
std::endl;

    return 0;
}
#endif
```

The expected output.

```
[Starting history]:
========== BEGIN TRANSACTION HISTORY ============
0   VGRO    10  1   2018    Buy 150 10300.14    0.00    0   0.000   0.000
1   VGRO    24  2   2018    Buy 85  7423.05     0.00    0   0.000   0.000
2   VGRO    8   8   2018    Buy 43  3367.76     0.00    0   0.000   0.000
3   VGRO    11  11  2018    Buy 78  7028.58     0.00    0   0.000   0.000
4   VGRO    8   12  2018    Sell    55  5958.15     0.00    0   0.000   0.000
5   VGRO    22  12  2018    Sell    80  2817.60     0.00    0   0.000   0.000
6   VGRO    4   1   2019    Buy 65  3257.15     0.00    0   0.000   0.000
7   VGRO    7   5   2019    Buy 65  4557.15     0.00    0   0.000   0.000
8   VGRO    14  6   2019    Sell    80  4451.20     0.00    0   0.000   0.000
9   VGRO    16  7   2019    Buy 25  1752.75     0.00    0   0.000   0.000
10  VGRO    19  7   2019    Sell    90  6780.60     0.00    0   0.000   0.000
11  VGRO    20  10  2019    Buy 100 9011.00     0.00    0   0.000   0.000
========== END TRANSACTION HISTORY ============
[Sorted          ]:
========== BEGIN TRANSACTION HISTORY ============
0   VGRO    10  1   2018    Buy 150 10300.14    0.00    0   0.000   0.000
1   VGRO    24  2   2018    Buy 85  7423.05     0.00    0   0.000   0.000
2   VGRO    8   8   2018    Buy 43  3367.76     0.00    0   0.000   0.000
3   VGRO    11  11  2018    Buy 78  7028.58     0.00    0   0.000   0.000
4   VGRO    8   12  2018    Sell    55  5958.15     0.00    0   0.000   0.000
5   VGRO    22  12  2018    Sell    80  2817.60     0.00    0   0.000   0.000
6   VGRO    4   1   2019    Buy 65  3257.15     0.00    0   0.000   0.000
7   VGRO    7   5   2019    Buy 65  4557.15     0.00    0   0.000   0.000
8   VGRO    14  6   2019    Sell    80  4451.20     0.00    0   0.000   0.000
9   VGRO    16  7   2019    Buy 25  1752.75     0.00    0   0.000   0.000
10  VGRO    19  7   2019    Sell    90  6780.60     0.00    0   0.000   0.000
11  VGRO    20  10  2019    Buy 100 9011.00     0.00    0   0.000   0.000
========== END TRANSACTION HISTORY ============
========== BEGIN TRANSACTION HISTORY ============
0   VGRO    10  1   2018    Buy 150 10300.14    10300.14    150 68.668  0.000
1   VGRO    24  2   2018    Buy 85  7423.05     17723.19    235 75.418  0.000
2   VGRO    8   8   2018    Buy 43  3367.76     21090.95    278 75.867  0.000
3   VGRO    11  11  2018    Buy 78  7028.58     28119.53    356 78.987  0.000
4   VGRO    8   12  2018    Sell    55  5958.15     23775.22    301 78.987
1613.841
5   VGRO    22  12  2018    Sell    80  2817.60     17456.23    221 78.987
```

```
-3501.396
6   VGRO    4   1   2019    Buy 65  3257.15     20713.38    286 72.424  0.000
7   VGRO    7   5   2019    Buy 65  4557.15     25270.53    351 71.996  0.000
8   VGRO    14  6   2019    Sell    80  4451.20     19510.86    271 71.996
-1308.464
9   VGRO    16  7   2019    Buy 25  1752.75     21263.61    296 71.837  0.000
10  VGRO    19  7   2019    Sell    90  6780.60     14798.32    206 71.837
315.313
11  VGRO    20  10  2019    Buy 100 9011.00     23809.32    306 77.808  0.000
========== END TRANSACTION HISTORY ============
[CGL for 2018    ]: -1887.555
[CGL for 2019    ]: -993.151
```

10. Marmoset Submission Details

This project has *three* submissions. The submissions prior to the final submission are subsets of the final submission. Each student is required to submit to all parts. Refer to syllabus for late submission policies.

The purpose of requiring students to make multiple submissions is to assist students learn to tackle a larger scale codebase, which requires work over a longer period of time. With a single submission, students tend to leave too much coding to be done close to the final deadline, resulting in higher stress and ineffective learning. The multiple submissions also helps students to know which functions to focus on first.

First Submission

Project name: Project #4 First Submission Submission filename: History_Transaction_definitions.cpp

Due: **11:59pm on Friday, November 18, 2022**

- The methods (functions) tested in **First Submission** are the following:
    1. `Transaction::Transaction()`
    2. `Transaction::~Transaction()`
    3. `Transaction::operator<(Transaction const &other)`
- You are still required to provide **skeletons** ("do nothing definitions") for all member functions defined in Tasks (for both `Transaction` and `History`)
- The tests are out of 12, but we will mark it out of 8
    - ex: 9/12 will be marked as 9/8
- Which means:
    - Bonus: any grade above 8/12 will count as bonus toward the project total grade

Second Submission

Project name: Project #4 Second Submission Submission filename: History_Transaction_definitions.cpp

Due: **11:59pm on Friday, November 25, 2022**

- The methods (functions) tested in **Second Submission** are the following:
    1. `History::History()`(although not explicitly tested)
    2. `History::~History()`
    3. `History::read_history()`
    4. `History::insert()`
    5. `History::print()`
- You are still required to provide **skeletons** ("do nothing definitions") for all member functions defined in Tasks (for both `Transaction` and `History`)
- The tests are out of 15, but we will mark it out of 9
    - ex: 10/15 will be marked as 10/9
- Which means:
    - Bonus: any grade above 9/15 will count as bonus toward the project total grade

Final Submission

Project name: Project #4 Final Submission Submission filename: History_Transaction_definitions.cpp

Due: **11:59pm on Friday December 2, 2022**

- Your final grade will be
    - The sum of *Project #4 First Submission* out of 5 AND *Project #4 Final Submission*
    - The sum of
        - *Project #4 First Submission* out of 8
        - *Project #4 Second Submission* out of 9
        - *Project #4 Final Submission*

The overall project grade will be capped at 100%. It is not possible to earn a grade higher than 100% on the overall project grade.

Main Definition

If you submit your `History_Transaction_definitions.cpp` with a `main` function, please wrap it with the preprocessor directive`#ifndef MARMOSET_TESTING ... #endif`.

Plagiarism detection software

We analyze all submissions with automated plagiarism detection software. Please consult the syllabus for more information.