CSCE 121-517,518,519,520: Introduction to Program Design and Concepts
Spring 2017
Project

Phase-1 Project Due Sunday, Apr 9, 11:59 PM.
Phase-2 (Final) Project Due Sunday, Apr 30, 11:59 PM.
Review the Project and Program General Instructions!

---

1. **Phase-1, Brokerage Class:** Write code for a **Brokerage** class which need only deal in **USD** (US dollars) for cash. To support this **Brokerage** class you will also need the following:
   - A **Stock** struct to hold information about a particular **Stock** (a string for the **Stock** type, e.g. "INTC", and a double for the **Stock** buy/sell price in USD, e.g. 35.91 USD's). We will only deal with **Stock** of the following 5 types (with prices from 03/01/2017):

     - `Intel:  "INTC",  buy-sell-price   35.91/USD`
     - `Google: "GOOG",  buy-sell-price  830.63/USD`
     - `Apple:  "AAPL",  buy-sell-price  138.96/USD`
     - `Yahoo:  "YHOO",  buy-sell-price   45.94/USD`
     - `IBM:    "IBM",   buy-sell-price  180.53/USD`

   - A **Shares** class to hold information about the stock shares objects in the **Brokerage**, i.e. the shares **Stock** type, the number of shares (a double so that we can have fractional shares) of that **Stock** type, and the equivalent USD value (also a double) of that number of shares. These **Shares** objects should be used to track: 1) the total stock **Shares** (and total USD equivalent value) held by the **Brokerage**, 2) the stock **Shares** (and USD equivalent value) held by any customer (**Patron**) of the **Brokerage** in their account, and 3) the stock **Shares** (and USD equivalent value) involved in any transaction done by **Patrons** of the **Brokerage** (i.e. stock **Shares** Buys and Sells).
   - A **Patron** class to hold information about the customers in the **Brokerage**, i.e. their name (a string), their account number (an int), their various stock **Shares** held (and USD equivalent value), their cash account in USD's (a double), and their account balance in USD's (also a double, the total of stock **Shares** USD equivalent value and cash account USD value). To avoid difficulties with reading into a string variable, use the underscore character (_) in place of spaces for **Patron** names.
   - A **Transaction** struct to hold information about **Patron** stock **Shares** Sale/Buy in their **Brokerage** account, and **Patron** cash Add/Remove to/from their **Brokerage** account. Each **Transaction** should include: 1) Stock **Shares** or cash involved in transaction, 2) the **Patron** name, account number and resulting account balance in terms of stock **Shares** value and cash in USD's, 3) a string containing the transaction type (i.e. stock **Shares** Sale or Buy, cash Add or Remove), and 4) the transaction date (using **Chrono::Date**) and time (using **Chrono::Time**).
   - The **Brokerage** class should contain a list (vector) of **Patron**s and a list (vector) of **Transaction**s done by the **Patron**s.
   - The complete **Brokerage** state (name, **Stock** prices, stock **Share**s, cash in USDs, **Patron**s, **Transaction**s) should be able to be initialized from a text file (assuming the **Brokerage** name in the text file matches the current **Brokerage** name), and to be saved to a text file. When the main program first constructs a **Brokerage** (and gives it a name) the initialize from file option should be presented to the user, and if the user declines then the **Brokerage** should be initialized to a state with no stock **Shares** or cash or **Patron**s or **Transaction**s. When the main program exits (by user Menu choice) the save **Brokerage** to file option should be presented to the user, and if the user declines then no state is saved.

- **Brokerage** member/helper functions should be used to appropriately implement the main() program user Menu operations listed below.

Your **BrokerageMain** main program should consist of a user Menu loop that repeatedly queries the user for what to do next. Menu options must include:

- displaying the **Brokerage** state (stock **Shares** amount & USD value, cash USD value, total USD value),
- adding a new **Patron** to the **Brokerage**,
- checking whether someone is already a **Patron**, and displaying their information,
- displaying a list of information about all **Patron**s,
- adding cash (in USD) by a **Patron**,
- removing cash (in USD) by a **Patron**,
- doing a stock **Shares** sale by a **Patron**,
- doing a stock **Shares** buy by a **Patron**,
- displaying a list of **Patron**s that are overdrawn in their cash account (deadbeats)
- displaying a list of all **Transaction**s done by all **Patron**s
- quitting the program

Menu item details:

- When displaying the **Brokerage** state the user should additionally be asked if they wish to see the state for 1) all **Shares** of a particular one of the five **Stock** types, 2) for all **Cash** accounts, or 3) for ALL (i.e. all **Shares** of all five **Stock** types and all **Cash** accounts and the total USD value all **Stock Shares** and all **Cash**). Note that at any point in time the **Brokerage** state matches the sum of all assets held in **Patron** accounts.
- When adding a new **Patron** if the account number matches the number of an existing **Patron** then the new **Patron** must NOT be added. When adding a new **Patron** if the account name matches the name of an existing **Patron** then the new **Patron** must STILL be added (i.e. a **Patron** can have multiple accounts in the same **Brokerage**). All **Transactions** are based upon **Patron** account number NOT upon **Patron** account name.
- Displaying all **Patrons** results in no display if there are no **Patrons**.
- Adding **Cash** to a **Patron** account adds it to the **Patrons Cash** account.
- Removing **Cash** from a **Patron** account removes it from the **Patrons Cash** account. Any attempt to remove more **Cash** than currently held in the **Patrons Cash** account must be refused.
- Doing a stock **Shares** sale removes the **Shares** from the **Patrons** account and then adds the equivalent USDs to the **Patrons Cash** account. Note that if the **Patron** does not currently have a sufficient number of **Shares** of the **Stock** type in their account then the sale must be refused.
- Doing a stock **Shares** buy adds the **Shares** to the **Patrons** account and then removes the equivalent USDs from the **Patrons Cash** account. Note that stock **Shares** buys must never be refused even if it results in the **Patron Cash** account being overdrawn (in fact this is the only way a **Patron** can become a **"deadbeat"**).
- Displaying all **"deadbeats"** results in no display if there are no **"deadbeats"**.
- Displaying all **Transactions** results in no display if there are no **Transactions**.

To successfully complete **Phase-1** you must turn-in a main program **BrokerageMain.cpp** and all supporting files: **Brokerage.h, Brokerage.cpp, Chrono.h, Chrono.cpp and std_lib_facilities_4.h**, plus the special Project Cover Sheet and a standard testing doc. Your **Phase-1** program will be tested (per the Project Rubric) to ensure it can correctly perform all of the above specified user Menu operations. All appropriate error conditions will also be tested for correct handling.

2. **Phase-2 (Final), Brokerage Network:** Starting from the Phase-1 code fully replace the original console Menu interface with a Graphics Window GUI interface. Widgets of type Button, In_Box, Out_Box, Menu, etc should be used to completely replace the original **BrokerageMain** Menu operations implemented in **Brokerage** member/helper functions. The one allowable exception to this is that the initial "load from file" and the final "save to file" interface can continue to be thru the command line interface in the Unix/Linux window (on build.tamu.edu) the program was run from.

   Now write a new main() program **BrokerageMainGUI** that constructs three **Brokerage**s, each with a unique **Brokerage** name. Each of these **Brokerage**s must still support all five **Stock** types (using the above exchange rates from 03/01/2017). When the new main() program starts up each of the three **Brokerage**s (as they are constructed) will do their own "initialize from file" query (therefore each **Brokerage**s state should be initialized-from/saved-to a different file), and then pop-up their Graphics/GUI Interface Window (so three windows will be opened, one per **Brokerage**). Each **Brokerage** window can be independently used/closed (no need to close all three at the same time), and upon closing each **Brokerage** will do their own "save to file" query.

   An additional operation should be added to the **Brokerage** GUI, which is the ability for a **Patron** to transfer cash (in USD) from their account in one **Brokerage** to their account in a different **Brokerage**. The **Patron** must have accounts in both **Brokerage**s and sufficient cash in the "from" **Brokerage** to cover the transfer (i.e. the transfer should be rejected if it causes the **Patron** to be overdrawn in the "from" **Brokerage**). The transfer must always be initiated in the GUI window of the "from" **Brokerage**. The result of the transfer will not be visible in the window of the "to" **Brokerage** until some operation is done in the "to" **Brokerage** window which causes it to be redrawn.

   Your **Phase-2 Final** program will be tested to ensure that all three **Brokerage**s work independently and correctly (via their individual GUI Windows), and that transfers can be successfully made between the **Brokerage**s (again via their GUI Windows). To successfully complete **Phase-2** you must turn-in a main program **BrokerageMainGUI.cpp** and all supporting files: **BrokerageGUI.h, BrokerageGUI.cpp, Chrono.h, Chrono.cpp, Graph.h, Graph.cpp, GUI.h, GUI.cpp, Point.h, Simple_window.h, Simple_window.cpp, Window.h, Window.cpp and std_lib_facilities_4.h**, plus the special Project Cover Sheet and a standard testing doc. Your **Phase-2** program will be tested (per the Project Rubric) to ensure it can correctly perform all of the previously specified user Menu operations (now implemented in the GUI) and the new Transfer operation. All appropriate error conditions will also be tested for correct handling.

Phase-1 of this Project is worth **70%** of the full Project grade and must be turned-in to eCampus by **11:59pm on Sunday Apr 9, 2017**. **Phase-2** of this Project (i.e. the **Final** version of the Project) is worth **30%** of the full Project grade and must be turned-in to eCampus by **11:59pm on Sunday Apr 30, 2017**. The Project grading rubric (not the Homework Program grading rubic) will be used to Grade each Phase of the Project.

Since the expectation is that Project Teams will spend part of each Lab working together on the Project, Lab attendance will be **MANDATORY** from 3/28/17 thru 4/27/17. Each unexcused Lab absence will cost the Student 1% of the total points available for the Project (note that this deduction is just on the individual Student, not on the other members of their Project Team). The TA's will track attendance at each Lab and it will be their decision as to whether or not to excuse an absence.