Coin-Operated Software

Due Tuesday, February 18 at 8 a.m.

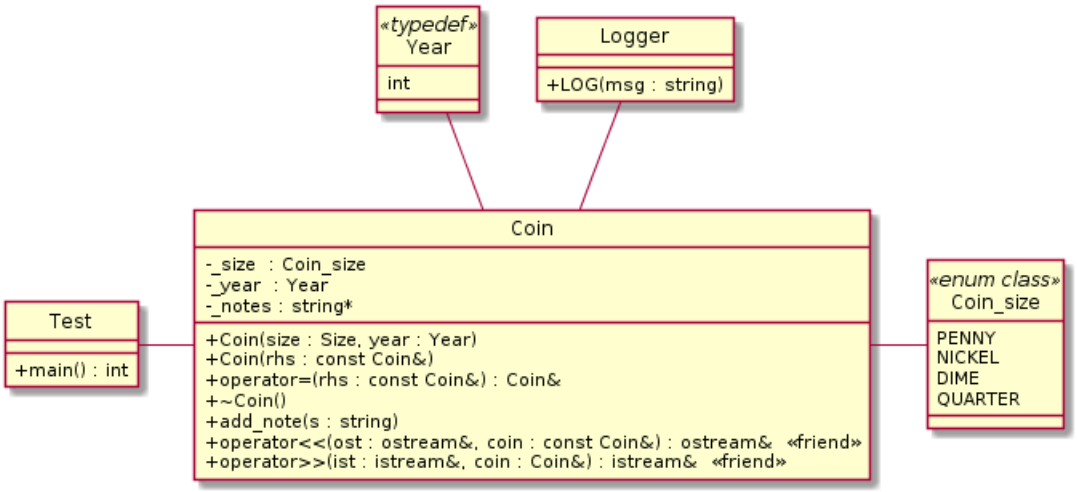
CSE 1325 - Spring 2020 - Homework #4 - 1 - Rev 2

Assignment Overview

Now that we've discovered the Rule of 3 - if you need a destructor, copy constructor, or copy assignment operator, you likely need all 3 - let's put them to good use! We'll model a simple US coin in 4 sizes - penny ($0.01), nickel ($0.05), dime ($0.10), or quarter ($0.25), with the year it was minted and some associated notes (such as its condition and history), as a numismatist (coin collector) might want when cataloging the coins in their collection.

Full Credit

In your git-managed Ubuntu Linux 18.04 directory **cse1325/P04/full_credit**, write a Coin class matching the following UML class diagram. IMPORTANT: Note that test.cpp and logger.h are provided at **cse1325-prof/P04/full_credit** for your use.



Note that Coin_size is an enum *class*, not a plain enum, and serves as the type for Coin::_size. **Overload operator<< for Coin_size**, streaming "penny", "nickel", "dime", and "quarter" for the respective values, and throwing a runtime error for any other value. (No other value is *currently* possible, but you never know when someone might add "DOLLAR" and forget to cover that in operator<<!)

Typedef Year to be an int to serve as the (more readable) type for Coin::_year. You needn't validate the year, but if you insist, the first US coins were minted in 1792. Over-achievers can look up the first date for each supported coin size.

The _notes pointer should be initialized to nullptr (that is, pointing to nothing). When add_note is called, if _notes is still nullptr, allocate a string on the heap (i.e., use the new keyword) and copy the parameter there, otherwise just append the parameter to the existing _notes string.

The executable members of Coin are as follows:

- The **constructor** should initialize _size and _year to the matching parameters, and _notes to nullptr. LOG the message "Coin::Coin" to help you visualize when special members are executed.

- The **copy constructor** should construct _size and _year from the matching rhs attributes, but make a "deep copy" of rhs._notes. That is, allocate a new string on the heap containing the same text as pointed to by rhs._notes. LOG the message "Coin::Coin copy constructor" to help you visualize when special members are executed.
- The **copy assignment operator** should copy across the _size and _year, but make a "deep copy" of _notes. That is, allocate a new string on the heap containing the same text as rhs._notes. LOG the message "Coin::operator=" to help you visualize when special members are executed.
- The **destructor** should delete _notes. LOG the message "Coin::~Coin" to help you visualize when special members are executed. (Hint: Deleting a pointer containing nullptr is guaranteed to be safe, so you should NEVER need to check for nullptr before deleting.)
- **Coin::add_note** should allocate a new string on the heap containing the parameter if _notes is nullptr (Hint: The idiom is `if(!_notes) _notes = new...`), otherwise, just append the parameter to the existing notes.
- `ostream& operator<<(ostream& ost, const Coin& coin)` will stream out the year, a space, the date, a newline, and then the notes.

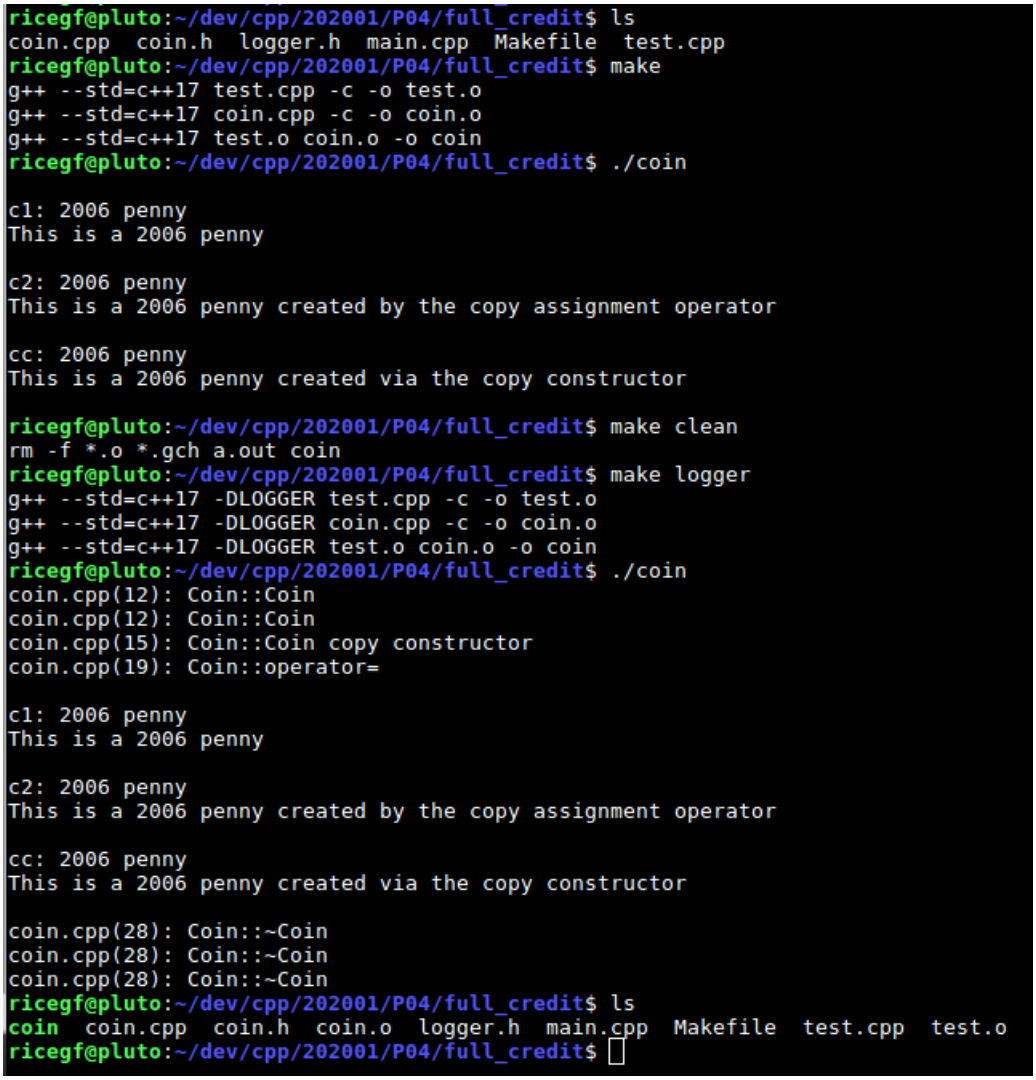
A test main is provided to (lightly) verify that your code works as specified. It should produce the output shown on the next page.

You MUST write a valid Makefile (as covered in Lecture 04) such that

- `make clean` deletes (at a minimum) your .o and .gch files and your executable(s).
- `make` builds the test main into an executable, with separate compilation of test.cpp and coin.cpp initiated only when required.
- `make logger` enables the logger.h LOG messages. (Hint: The compiler option to enable logging is -DLOGGER. You may find it helpful to model the `logger` target after the `debug` target in Lecture 04, slide 38.)
- `make test.o` compiles test.cpp into test.o only if test.cpp is newer than test.o.
- `make coin.o` compiles coin.cpp into coin.o only if coin.cpp is newer than coin.o.

Add, commit, and push all files.

Example Full Credit Output



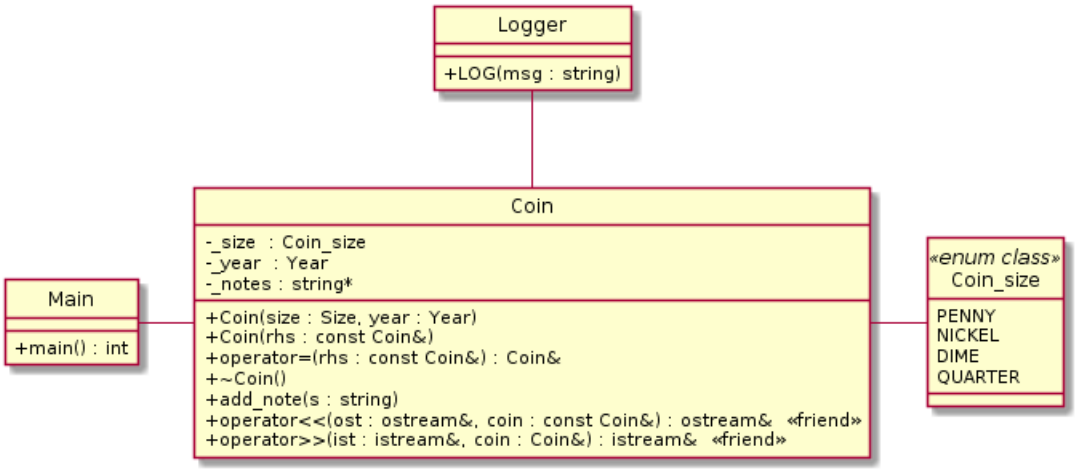
Bonus

Add a streaming in operator>> for both Coin_size and Coin. Throw a runtime error exception if bad input is provided.

Then implement a menu-driven application that manages a vector of Coin objects, with the following menu options:

- **Add a coin** - Use Coin's operator>> to obtain the coin from the user, and push it to the back of the vector.
- **List all coins** - List each coin's vector index, a close parenthesis,and the output of that coin's operator<< (the notes may be multi-line).
- **Delete a coin** - Ask for the index number of the coin to delete, and erase it. (Hint: The C++ idiom to erase element `index` of vector `v` is `v.erase(v.begin() + index);` - something like v.begin() is providing a pointer to the start of the vector, and so you do some pointer math. More or less.)
- **Exit** - All data is lost, since we haven't yet learned how to do file I/O. But it's coming after the first exam!

IMPORTANT: Do NOT abort! Catch any thrown exceptions (i.e., you'll need a try / catch clause in your main() function), report each exception to the user, and then return to the menu for the next command.



Add, commit, and push all files.

```
ricegf@pluto:~/dev/cpp/202001/P04/bonus$ make
g++ --std=c++17 main.cpp -c -o main.o
g++ --std=c++17 coin.cpp -c -o coin.o
g++ --std=c++17 main.o coin.o -o coin
ricegf@pluto:~/dev/cpp/202001/P04/bonus$ ./coin

===============
 CSE1325 COINS
===============
A)dd a coin
L)ist all coins
D)elete a coin

Q)uit

0 coins >> a
Coin (year size \n notes): 2020 penny
Quite a pretty penny!

===============
 CSE1325 COINS
===============
A)dd a coin
L)ist all coins
D)elete a coin

Q)uit

1 coins >> l

oooooooooooooo
   C O I N S
oooooooooooooo

0) 2020 penny
Quite a pretty penny!
```

Extreme Bonus

Using the existing operator>> and operator<< definitions in Coin, add 2 additional commands to your user interface:

- **Save all coins** - Ask the user for the filename to which to save the coins. If the user just hits Enter, then use the most recently opened or saved filename (or "untitled.coins" if this is the first save). Open the specified filename for output *using C++ classes* (no fopen for you!), and stream the coins into the file. Catch and handle any exceptions thrown during the save.
- **Open a coin file** - Ask the user for the filename to open. If the user just hits Enter, then abort the open. Otherwise, open the specified filename for input *using C++ classes*, clear the coins vector, and stream the coins from the file into the vector. Note that the file to be opened must be the same file as was saved above! Catch and handle any exceptions thrown during the open.

Consider (or test) what happens if you try to open a *non-coin* file. Will your program recognize that the file wasn't written by your program immediately, or will it flounder about trying to load gibberish? Also consider what will happen if the file format changes. How will your program determine if the file format is of the old or new variety? You're not required to *solve* these challenges (necessarily), but it's the kind of issues that professionals consider. Perhaps you should, too.

And like all professionals - add, commit, and push all files.

```
ricegf@pluto:~/dev/cpp/202001/P04/extreme_bonus$ make
g++ --std=c++17 main.cpp -c -o main.o
g++ --std=c++17 coin.cpp -c -o coin.o
g++ --std=c++17 main.o coin.o -o coin
ricegf@pluto:~/dev/cpp/202001/P04/extreme_bonus$ ./coin

===============
 CSE1325 COINS
===============
A)dd a coin
L)ist all coins
D)elete a coin

S)ave all coins
O)pen coin file

Q)uit

0 coins >> o
Filename? untitled.coins

===============
 CSE1325 COINS
===============
A)dd a coin
L)ist all coins
D)elete a coin

S)ave all coins
O)pen coin file

Q)uit

3 coins >> l

ooooooooooooooo
   C O I N S
ooooooooooooooo

0) 1992 penny
What a pretty penny!

1) 1902 nickel
Old and valuable!

2) 2020 dime
Bright and shiny!
```