CISP 400 - Lab Assignment #8
Due Thursday, November 1, 2018 at 11:59 PM

This program is intended to give you a chance to work with more advanced features involving inheritance, and to use smart pointers, specifically unique_ptr and shared_ptr. These are very complicated topics and using them in their entirety might be several lab projects by themselves, but it is intended to give you an overview and feel for how to write a program that functions well using these features.

The overall program involves a hypothetical bank computer system that handles paying off debts. It will start with an account that is in debt, and the user will input payments, either by credit card or check, until the debt is entirely paid off or the user decides to exit.

There will be four classes: Payment, CheckPayment, CreditCardPayment, and Account. CheckPayment and CreditCardPayment will be child classes of Payment, which will not be instantiated in and of itself. Account will not have parent or child classes.

<u>Account Class:</u>

Account keeps track of the current debt as a double. It keeps a private member variable that tracks how much of the current debt remains. It also has two private variables that are each shared pointers that point to the most recently applied check payment and credit card payment. These variables have a value of nullptr if there have been no credit card payments or check payments applied, and they should be set to nullptr in the constructor.

In terms of actual code, Account has several public functions. One of them, stillInDebt returns a bool, and tells any calling function whether there is still a positive debt to pay off. Another reports the current debt. There are also getters that provide shared_ptr objects, which own the CheckPayment and CreditCardPayment objects most recently used to reduce the debt in Account.

Account has several friends, including a function to apply a credit card payment, a function to apply a check payment, and an output overloaded operator <<.

The overloaded operator will display the status of the account. It will do this by displaying the current debt. It will also display the most recent CheckPayment <u>and</u> most recent CreditCardPayment. It must display both. If no CreditCardPayment or CheckPayment has been applied to the account yet (in which case these shared pointers should have a value of nullptr), this must be indicated instead of printing out the data on the payment.

It has two **<u>protected</u>** member overloaded operator functions for -=. One of them will take a Payment parameter, and the other will take a CreditCardPayment parameter. Either way, they will subtract the amount in the payment from the current debt. They will each be used in friend functions (one for CheckPayment, one for CreditCardPayment) that apply the payment using -=, and also set the appropriate shared pointer that tracks the most recent credit card or check payment. Note that these two protected member overloaded operator functions can be accessed by friends.

Account also has a constructor that initializes the amount of debt with a double parameter, and sets the shared pointers to null. The destructor will not clean or deallocate anything. However, it will check the current debt amount. If the current debt amount is greater than 0, it will print out a message to the user, warning them that the debt is not yet settled. If the current debt is less than or equal to 0, it will print out a message to the user noting that the debt has been paid off. Remember you have a member function that can do much of this calculation by itself. NOTE: The Account does have dynamically allocated memory that holds up to two objects, the most recently applied CheckPayment and CreditCardPayment objects (classes described in more detail below). However, if you have been using smart pointers correctly, these will AUTOMATICALLY self-destruct once they go out of scope, so you should not have to do anything special to ensure that they will be properly disposed of. In fact, you should not have to do anything special to make sure that the Account object is properly disposed of either, since it is owned by a unique_ptr, which is itself a smart pointer that should not go out of scope until the program exits.

Payment Class:

Payment objects track payments, accepting the name of the payer, and the amount that they are for. They are intended to be applied to Account objects, and over time eventually apply enough for the debt quantity in Account to end up at 0 or a negative balance (overpayment).

Payment will have a protected member, getNamedPayer. This will be used by child classes to access the name of the payer.

Payment will also have a member function named getAmount() that acts as a getter to let other parts of the program know how much money is in that particular payment.

Payment objects should have a destroyer that announces that a Payment object is being destroyed.

CreditCardPayment Class:

This is a child class of Payment. In addition to the default data tracked by Payment objects, this will also carry data on the credit card number, and will also make some modifications and recalculations to other data obtained from these objects that would not be available in standard Payment objects. NOTE: The credit card numbers are SIMPLE INT VARIABLES. We are not trying to have realistic credit card numbers here.

The class has a public member, getName(), to get the name of the payer. This will take the name of the payer from the base class, and append the string “ (by credit card)”. It will also have a public getter function, getCardNumber(), which gets the credit card number.

This class will override getAmount(). This is because it will take the results of getAmount() from the base class (Payment) and then calculate it to remove 10% of the total amount. This is to simulate the credit card company taking a (very large!) portion of the transaction as payment for their services. This amount should be specified by a constant.

This class has an overloaded operator that is NOT a member, but is a non-member friend. It is the overloaded operator <<, and allows you to print out the contents to cout or another file. It should print the name of the payer (through its own getName() method, NOT through getNamedPayer() from its parent), the amount of money in the payment (using its own getAmount(), NOT through the parent version), and the credit card number.

CreditCardPayment objects should have a destructor function that announces a CreditCardPayment object is being destroyed. Note that this destroyer will execute about the same time as the object's Payment destructor function, so you will get two outputs every time a CheckPayment is destroyed.

CheckPayment Class:

This is a child class of Payment.

CheckPayment objects should have a destructor function that announces a CheckPayment object is being destroyed. Note that this destructor will execute about the same time as the object's Payment destructor function, so you will get two outputs every time a CheckPayment is destroyed.

CheckPayment objects hold one additional piece of information that a Payment object does not – the name of a co-signer. A co-signer is usually a person who will pay in a dispute if the original person is unwilling to pay. Co-signers are not typically used on checks, but this is not intended to be a fully realistic simulation.

CheckPayment has a function, getName, that gets the name of the payer. Not unlike the version for Credit Card, it should use the base class method to get the name of the payer. It should also append the string, " (by check)" to the name before it returns it to the calling function.

Like CreditCardPayment, CheckPayment also has an overloaded friend operator, <<, which reports the name, amount, and co-signer in the payment.

MAIN PROGRAM:

The very first thing that the main program will do is get the amount of the debt from the user, in dollars and cents. Once this has happened, the program use it to create an Account object, which is owned by a unique_ptr.

Then we get into a menu loop. The options presented to the user are to submit a new checking payment, submit a new credit card payment, view the current account status, and exit. The menu will check to make certain that the input is valid. If it is valid, it will act on it. If not, it will continue to ask the user for input until it gets a valid response.

If the user selects to exit, the program exits the menu loop. The menu loop is ALSO exited when the Account is checked to see if it is still in debt (remember, it should have a member function to do that calculation, to determine whether or not the account is still in debt). Either way, once the menu loop is exited, the computer uses << to print the Account for one final report. Then the main() function exits. Note that there should be one last output by the Account object as it is destroyed (which should indicate whether the Account was in debt or not at the time of its destruction).

NOTE: The program will, every iteration, CHECK to see if the debt is less than or equal to zero. The loop will AUTOMATICALLY END if the debt has been completely paid off.

Payment Creation Utility Functions:

There will be two object creation utility functions, one for each subclass of Payment. Each will query the user about every value in that particular type of Payment (e.g. name, amount, credit card number, etc.). The appropriate type of object will be constructed using a shared_ptr, and then returned.

The program will afterwards feed the shared_ptr object into the appropriate payment application function, which is a friend of Account.

NOTE that you MUST have two different ‘apply payment’ functions, one for each kind of payment. This is because CreditCardPayment recalculates getAmount() in order to account for how much money the credit card company takes, and because the Account tracks the most recent credit card payment, and the most recent check payment.

Apply Payment Functions:

These have been mentioned before. There will be two of these. They will be FRIENDS of the Account class. Each one will take two parameters. Both will have one parameter that is a REFERENCE to a unique pointer for an Account. The second parameter will differ between the two. One will be a shared pointer to a CreditCardPayment object, and the other will be a shared pointer to a CheckPayment object. However, the objective of these is the same. They will both use the overloaded -= operator with Account and their respective permutation of Payment to reduce the debt by the appropriate amount, and then set themselves as the most recent payment of their respective type on the account.

TIPS:

- DO NOT HAVE THE CONSTRUCTOR OF THE OBJECTS TAKE USER INPUT. This limits their reusability greatly. This mistake has arisen several times in grading. It is generally a poor design decision; you should usually have what you need to build the object, if at all possible, BEFORE you build it. So collect the data first, and then feed it to the object constructor.
- On output, the credit card will display the value AFTER the amount that the credit card companies takes. So if you put in $1.00, the amount of money that you remove from the debt is $.90. Status reports on the object will result in the amount appearing to be $.90.
- You can assume all money values are in dollars and cents. Just put in the plain number, don’t bother with a $ sign.
- You do not have to do any input validation outside of the main menu. This means you can do silly things, such as negative payments (that would actually increase the debt). The point of this exercise is to work with smart pointers and inheritance, not do validation checking. Validation checking is important, but you will be busy enough with the program as-is.
- ORGANIZE YOUR PROGRAM. If you use one file, then group together the implementations of the different classes. If you use multiple files, group the classes in their own implementation files, and have an additional file that contains the functions that are not part of any class. Organize within those, as well. Having a program with functions that are all over the place is bad coding practice and may cost you points. Use forward declaration as needed.
- All classes must have constructors. They must ALSO call the constructors of their parents (if they have any) in the initializer list. All classes must also have destructors that announce that the object is being destroyed.
- Do NOT overwrite, or do any kind of adjustment, for setting the amount in CreditCardPayment. The object should STILL store the original amount the user put into the computer. This is in case someone wants to write a program with the object later, and needs to access the original amount that was input.

- Be sure to have the -= overloaded operator be separate from the function that applies the payments to the Account. The function that applies the payments to the account should use the -=, and should also set the appropriate ‘most recently submitted’ payment on the Account object. Furthermore, the apply-payment functions should accept a CheckPayment and CreditCardPayment (which is what differentiates them), not a “plain” Payment object. The operator -=, on the other hand, should only subtract the relevant quantity. This means that there should be two of them – one coded to accept Payment, which will use the default Payment getAmount(), and another that is prototyped specifically to accept CreditCardPayment, which will use the overridden getAmount() for credit cards.
- You are using a lot of dynamic memory allocation in this program. Remember though that you should be concentrating on using smart pointers correctly, so that they can handle the heavy lifting, e.g. destroying objects that are no longer useful.

<u>Deliverables:</u>

Submit the program file or files to Canvas by the due date. Remember to include a header file if you use multiple files, and remember to include it with your code.