CS 142 Final Exam

Version 0.8

Instructors: R. P. Burton, K. Seppi, and E. Child

Monday, December 16, 2013 thru Friday, December 20, 2013, 8:00 p.m.

We will not be accepting late finals.

Allowed materials include your CS 142 course text, your own notes, your own prior lab solutions, the CS 142 course website, and cplusplus.com

Disallowed materials include all other text resources, all other Internet

resources, and all neighbors (remember, everyone is thy neighbor).

**Instructions**

1. This midterm consists of a C++ programming problem. Read and understand the statement of the problem completely before beginning to design, code, and test. Understanding the problem correctly is part of the examination. If something seems unclear, ask a CS 142 TA (but no one else) for clarification.
2. As part of your design, consider test cases that will establish the correctness of your solution.
3. Produce a solution, which consists of *your* C++ code, with a comment at the beginning of your file which includes your name, your student ID number, and “CS 142 Fall 2013 Final Exam.” Save your complete source file(s) using the online submission script. Attribute any code taken from or based on other sources (except for the course text and the course website). Attributed code taken from or based heavily on other sources is worth half credit. Unattributed code taken from or based heavily on other sources is worth no credit.
4. You may pose questions to the CS 142 TAs at any time. However, the TAs generally are not permitted to answer questions related to design, C++ implementation, debugging, or testing.
5. Code well. This includes choosing good names for identifiers, avoiding magic numbers, formatting in a communicative way so that your code is reasonably self-documenting, choosing good control structures, using appropriate data structures, encapsulating, using inheritance and polymorphism appropriately, etc. Provide comments where required or appropriate, especially to help the TAs understand what your code is intended to do. Remember, you will be graded not only on whether your code (when entered, compiled, and executed) accomplishes the specified task, but also on how clearly and efficiently it accomplishes the specified task, and on how closely you adhere to the stated requirements and to good programming practices.
6. When you are finished, score your solution as indicated on the Scoring Sheet. If you are within 15% of your score, you will not lose or gain extra points. If you are at within 16% to 25% of your score, you will lose 2.5% of your score. If you are not within at least 25% of your score, you will lose 5% of your score. After you complete the grading sheet, go to Learning Suite and follow the instructions under the Contents Tab in Exam Submission.
7. Sign here (1) to certify that you reviewed all your posted scores for the semester, notifying a CS 142 TA in writing to any discrepancies, (2) to request that your midterm be graded, and (3) to certify that no unfair information related to the midterm has been received by you, either directly or indirectly, and that none has been or will be conveyed by you. If we discover that you cheated or assisted someone in cheating, intentionally or unintentionally (including accidentally), your score for this exam may (and probably will) be rand() % 1. Discipline also may include Honor Code Office involvement, which can lead to loss of opportunities at Brigham Young University.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ 2013

(Name) (Date)

**You Can Bank on It**

**Problem Statement**

The bank at which you have been building up your resources has discovered your prowess as a C++ programmer and has hired you to automate much of their operation. Specifically, the bank wants you to write a C++ program to automate opening bank accounts, depositing into and withdrawing from existing bank accounts, closing bank accounts, and applying monthly fees and interest.

**Background:**

You’ve learned everything you need to know about the banking operation:

* A “bank account” is merely an abstraction that functions meaningfully as the root of a multi-level inheritance hierarchy.
* Any bank account allows deposits at any time and can be closed at any time (provided it has a nonnegative balance).
* A bank account is either some variety of checking account or some variety of savings account.
  + A Regular checking account can have any nonnegative balance, and incurs a monthly fee of $6.00.
  + A Regular+ checking account must maintain a balance of at least $300 and has no monthly fee; if the balance drops below $300, the monthly $6.00 fee is imposed.
  + A Regular++ checking account must maintain a balance of at least $800, has no monthly fee, and earns interest at 0.5% per annum. If the balance drops below $800, no monthly interest is posted. If the balance drops below $300, the monthly $6.00 fee is imposed.
  + A Regular savings account can have any nonnegative balance, has no monthly fee, and earns interest at 1% per annum.
  + A Regular+ savings account must have a balance of at least $1000 and earns interest at 1.25% per annum; if the balance drops below $1000, the interest rate drops to 1%.
  + Unlike other accounts which allow withdrawals at any time, the balance and accumulated interest in a CD savings account can be withdrawn only upon maturity of the account. It pays interest at 2% per annum. If a withdrawal is made prior to maturity, all interest earned on the remaining balance since the date of the withdrawal (and not already forfeited) is forfeited.
* If the balance is insufficient for the intended withdrawal (or the check), the withdrawal (or writing of the check) is disallowed and a $5.00 penalty is imposed (and debited from the account). If the balance in the account is less than $5.00, the penalty is the amount of the balance.
* All bank accounts have an account number, a current balance, and an owner. CD accounts also have durations (to maturity). To simplify the problem, you may assume that a CD account matures in 9 months. You may add additional data members to specific account classes.

**Requirements:**

All required methods are included in the interfaces discussed in the next section.

When the user opens an account, it must include an initial deposit. Account numbers are assigned, the number of the first account being 1, the number of the second account being 2, etc. Once the account is opened, the account number and the owner cannot be changed. *Instead of providing a member function to identify the type of the account, an object of the appropriate class is created and added to the collection of bank accounts.*

Deposits can be made by specifying the account number and the (positive) amount of the deposit.

Withdrawals can be made (or checks can be written) by specifying the account number and the (positive) amount of the withdrawal (or check), not to exceed the balance in the account (in which case a penalty is imposed).

An account may be closed provided its balance is nonnegative.

When the month advances, all applicable fees and interest must be applied to all accounts. Your program uses polymorphism (rather than if-statements or a switch-statement) to determine the type of each account, computing the current balance in a manner appropriate to the type of each account.

**Provided Files**

We have provided several files which you must use:

* Factory.h
* Factory.cpp
* BankInterface.h
* AccountInterface.h
* TestDriver.cpp
* (optional) Test1.txt

BankInterface.h

The BankInterface.h requires you to implement four different functions:

1. openAnAccount(string info)
2. closeAnAccount(int accountNumber)
3. getAnAccount(int accountNumber)
4. getNumberOfAccounts()
5. advanceMonth()

Instructions for what is required in each function are written in a comment block above the function declaration in the BankInterface.h file.

AccountInterface.h

The AccountInterface.h requires you to implement the following functions:

1. getAccountNumber()
2. getCurrentBalance()
3. getAccountOwner()
4. accountAdvanceMonth()
5. deposit()
6. withdrawFromSavings(double amount)
7. writeCheck(double amount)

Instructions for what is required in each function are written in a comment block above the function declaration in the AccountInterface.h file.

**Test Diver**

We have given you the test driver that we will be using to grade your final exam. This test driver, however, is **FILE DRIVEN**. This means that the test driver will read a file that tells it how to test. **We have given you an example of the kind of file the test driver reads, Test1.txt.**

The test driver must test Opening Accounts, Transactions, and Closing Accounts (in that order for the test driver).

When testing Opening Accounts, in the text file, each line must have:

[Account Type] [Balance] [Name] [true | false]

Each line in the file is what the test driver will read in. It will use the Account Type, Balance, and Name to pass it into your “openAnAccount(string info)” method in the BankInterface. The last part [true| false] is what the test driver expects the method to return. If the method does not return the right result, it will fail.

Example:

Opening Accounts:

Checking -1 Iris false // false: balance is nonpositive.

Checking 1000 Paul true

Checking+ -1 Sydney false

Checking+ 350 Adam true

Checking++ 700 Emily false // false: balance is insufficient for a Checking++ account.

Checking++ 1200 Michael true

Saving 40 Dantley true

Saving+ 3000 Jason true

Saving+ 200 Kevin false // false: balance is insufficient for a Saving+ account.

CD 400 Andrea true

Size: 6

When you are finished testing Opening Accounts, you must write:

Size: [Total Number of Accounts Expected]

Example from above:

Size: 6

When Testing Transactions, each line must have:

[Function Name] [Account ID] [[Parameter]] [[Result]]

(double brackets not applicable to all functions)

Example:

Testing Transactions:

writeCheck 1 1200 false

getCurrentBalance 1 995

advanceMonth

getCurrentBalance 1 989

writeCheck 1 300 true

getCurrentBalance 1 689

deposit 1 200

getCurrentBalance 1 889

withdrawFromSavings 5 1000 true

getCurrentBalance 5 2003

End

When you are finished Testing Transactions, you must write:

End

Example from above:

End

You may test any function in Testing Transactions in any order. Test1.txt is similar to what we will be using to test your solution, but less elaborate.

When testing Closing Accounts, you must write:

close [Account ID] [true|false]

Example:

Closing Accounts:

close 34 false

close 4 true

close 5 true

close 1000 false

Size: 4

When you are finished testing Closing Accounts, you must end with:

Size: [Total Number of Accounts Expected]

Example from above:

Size: 4

**IMPORTANT INFORMATION ABOUT TESTING:**

We will test numbers by rounding them to the nearest whole number. We will test with integers as seen in the example test file. In the test file, when you call advanceMonth() on an account, and interest is applied, you must apply interest using double arithmetic. But when we check to see if the interest was added correctly (by calling getCurrentBalance in the test file) we will expect a result that is rounded to the nearest whole number. See Figures 1 and 2. The Test Driver will round the actual result of getCurrentBalance() (originally must be a double), but you must write a rounded result in the text file. The TAs test files will do this during grading.

**Figure 1:**

advanceMonth

getCurrentBalance 13 3221 // 3221 is the result. It is rounded. This is correct.

**Figure 2:**

advanceMonth

getCurrentBalance 13 3221.3214353 // 3221 is the result. This is incorrect.

**Sample Interest Calculation:**

If the current balance is $150.25 and the interest rate is 1% per annum, the interest for one month is

$150.25 x 1%/year x 1 year/12 months = $0.128208333…,

being careful not to calculate 1year/12 months first, since both 1 and 12 are integers and 1/12 is 0.

**NOTE:**

* If you think you need to change the provided files (except as noted for the extra credit and possibly in the factory), you have done something wrong.
* As you have learned, files can be tricky; they can have extra lines that you are unaware of. If you have to add or delete a “getline” in the test driver for you test file, please note that on your grading sheet.

**Extra Credit:**  (5 points)

Implement “getSortedAccounts(string criterion)” in the BankInterface.

The method must return a vector<AccountInterface\*> sorted by a criterion. The criterion will be a string that is “accountNumber”, “name”, or “balance”. You must sort the accounts by that criterion. If the criterion is “accountNumber”, the vector must be sorted by increasing account numbers ( 1,2,3…). If it is “name”, it must be sorted by increasing ASCII value ( A,B,C,a,b,c…).

If the criteria is “balance”, the accounts must be sorted by increasing balance value

( 100,230,1200,...).

You must implement your own algorithm for the sort. Use of the C++ sort function will not be valid for extra credit.

**Final Scoring Sheet**

Name:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Section #:\_\_\_\_ Day Submitted: \_\_\_\_\_\_\_\_\_\_\_\_ T.A. Present \_\_\_\_\_\_\_\_

**Caution**: You will forfeit 10 points if your code has syntax errors and does not compile.

**Student TA Grading**

**\_\_\_\_/50 \_\_\_\_/50 Inheritance and Polymorphism**

\_\_\_\_/20 \_\_\_\_/ 20 Stores all Accounts in one vector in Bank

\_\_\_\_/20 \_\_\_\_/ 20 Account types represent individual classes

\_\_\_\_/10 \_\_\_\_/ 10 Account types inherit from one or more base classes

**\_\_\_\_/10 \_\_\_\_/10 Opening Appropriate Accounts**

\_\_\_\_/5 \_\_\_\_/ 5 Bank correctly opens accounts based on type of account

\_\_\_\_/5 \_\_\_\_/ 5 Bank correctly does NOT add invalid account information to Bank

**\_\_\_\_/15 \_\_\_\_/15 Appropriate Transactions**

\_\_\_\_/5 \_\_\_\_/ 5 Accounts correctly withdraws/writes checks or rejects & applies

penalties, all according to account type

\_\_\_\_/5 \_\_\_\_/ 5 Correctly advances month, adds/removes interest, adds appropriate

fees to accounts.

\_\_\_\_/5 \_\_\_\_/ 5 Accounts correctly deposit amounts.

**\_\_\_\_/10 \_\_\_\_/10 Closing Accounts Appropriately**

\_\_\_\_/5 \_\_\_\_/ 5 Can close accounts only if balance is non-negative

\_\_\_\_/5 \_\_\_\_/ 5 Closes accounts by removing from Bank

**\_\_\_\_/5 \_\_\_\_/5 Created and included a test file (must be different from Test1.txt)**

**\_\_\_\_/10 \_\_\_\_/10 Style and Organization**

\_\_\_\_/5 \_\_\_\_/5 .ccp and .h files are separated

\_\_\_\_/5 \_\_\_\_/5 Accounts hold appropriate variables

**\_\_\_\_/5 \_\_\_\_/ 5 Extra Credit: Implemented a self-written sorting algorithm.**

**\_\_\_\_/ 100 pts \_\_\_\_/ 100 pts TOTAL**

**\_\_\_\_/ 100 pts Adjusted TA Grader\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

*Student to TA* Comments:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*TA to Student* Comments:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_