Implement the "remove account" feature

• implement the remove account feature that removes an account based on the account number. You will implement the "remove account" feature, which is available from the admin menu. This feature prompts the user to enter an account number, and the corresponding account is removed from the list of bank accounts

Implement the transaction features

• You will implement the deposit and withdrawal features, which are available from the customer menu. Implement the deposit and withdrawal features, specifically the corresponding functions on the Account class; you do not need the transaction related classes

Your program will:

- prompt the user to enter the account number involved in the deposit or withdrawal
- prompt the user to specify the amount to be deposited or withdrawn
- perform the deposit or withdrawal on the corresponding Account object
- determine whether a deposit or withdrawal succeeded or failed; examples of failed transactions would be an attempt to deposit or withdraw a negative amount, or an attempt to withdraw an amount greater than the account balance

Other Implementations:

1. Create an account hierarchy You will create three new classes derived from the Account class: a savings account class (SavingsAcct), a chequing account class (ChequingAcct), and a general account class (GeneralAcct).

The SavingsAcct class will store and initialize two new data members: an interest rate and a penalty amount. The interest rate will be applied as a bonus to every amount deposited in the savings account. For example, if the interest rate is 10% and an amount deposited is $340 dollars, then the actual amount deposited will be $340 + 340*0.10 = $374. The penalty is a deduction imposed on every withdrawal from a savings account. For example, if the penalty is 5% and the amount to be withdrawn is $40, then the actual amount deducted will be $40 + 40*0.05 = $42. The ChequingAcct class will store and initialize one new data member: a cheque cost. The cheque cost is a deduction imposed on every withdrawal from a chequing account. For example, if the cheque cost is 50 cents and the amount to be withdrawn is $40, then the actual amount deducted will be $40 + $0.50 = $40.50. The chequing account does not provide a bonus for deposits.

The GeneralAcct class will store no new data members. The kind of account does not provide a bonus for deposits nor a penalty or deduction for withdrawals.

The Account class will now be abstract.

You will provide sufficient datafill for the different types of accounts. There will be no more Account objects in the program. This means that you must change the existing account initialization function so that a variety of savings, chequing and general accounts are created instead. You should still have a minimum of 15 accounts in total.

Note: Insufficient or incorrect datafill means that your assignment cannot be adequately tested; this may incur deductions of up to 100% of the assignment

2. Implement the polymorphic functions The deposit and withdraw functions on the account classes will be polymorphic, and they will perform the deposits and withdrawals as described above. You must implement these functions using polymorphism. Do not use RTTI and do not store the type of each object.

3. Implement the overloaded operators You will implement two overloaded operators on the account collection class (either AcctArray or AcctList):

Operator	Parameter	Functionality
+=	Account* acct	Add acct to this collection
-=	Int acctNum	Remove the account with acctNum from this collection

You will change your existing code to use these overloaded operators everywhere applicable.

Notes:

- o Your overloaded operators must reuse existing functions everywhere possible.
- o You must enable cascading everywhere appropriate.

Constraints

• your program must follow the existing design of the base code, including the encapsulation of control, UI, entity and array object functionality

• do not use any classes or containers from the C++ standard template library (STL)

• do not use any global variables or any global functions other than main

• do not use structs; use classes instead

• objects must always be passed by reference, not by value

• your classes must be thoroughly documented

Submission:

• a UML class diagram (as a PDF file) that you have drawn using a drawing package; the UML diagram must correspond to the design of the entire program

• one tar file that includes:

o all source and header files for your program

o a Makefile

o a readme file that includes:

- a preamble (program and modifications authors, purpose, list of source/header/data files)
- a preamble (program and modifications authors, purpose, list of source/header/data files)
- compilation, launching and operating instructions

Grading Marking

Components:

• UML diagram: 15%

15 marks: new associations

- 6 marks: correct relationship between account class and derived classes (2 marks each)
- 9 marks: correct attributes and operations on derived classes object (3 marks each)

Note: the UML must show the classes and associations for the entire program in order to receive marks

• Account derived class definitions: 30%

- 10 marks: correct attributes and operations on SavingsAcct class
- 10 marks: correct attributes and operations on ChequingAcct class
- 5 marks: correct attributes and operations on GeneralAcct class
- 5 marks: correct changes to Account class

• Polymorphic functions: 40%

- 20 marks: correct implementation of deposit and withdraw on SavingsAcct class (10 marks each)
- 10 marks: correct implementation of deposit and withdraw on ChequingAcct class (5 marks each)
- 10 marks: correct implementation of deposit and withdraw on GeneralAcct class (5 marks each)

• Overloaded operators: 15%

- 7 marks: correct implementation of += operator, including cascading
- 8 marks: correct implementation of -= operator, including cascading

Execution requirements:

• all marking components must be called, and they must execute successfully to receive marks

• all data handled by marking components must be printed to the screen for marking components to receive marks

Deductions:

• Packaging errors:

o 10 marks for missing Makefile

o 5 marks for missing readme

o 10 marks for consistent failure to correctly separate code into source and header files

o 10 marks for bad style or missing documentation

• Major programming and design errors:

o 50% of a marking component that uses global variables, global functions, or structs

o 50% of a marking component that consistently fails to use correct design principles

o 50% of a marking component that uses prohibited library classes or functions

- 50% of a marking component where unauthorized changes have been made to provided code
- up to 10 marks for memory leaks, depending on severity

- Execution errors:
  - 100% of a marking component that cannot be tested because the code does not compile or execute
  - 100% of a marking component that cannot be tested because the feature is not used in the code
  - 100% of a marking component that cannot be proven to run successfully because data is not printed out