CSP2104D

Object-Oriented Programming with C++

<u>Stage 7</u> Submission Due Date: 28th May, 2015 with demo on: 30th May, 2015 (in lab 2).

Related Learning Outcomes from Unit Outline:

On completion of this assignment students should be able to:

- formulate computer algorithms using the operations, control structures and classes provided in the C++ programming languages;
- write, test and debug computer programs written in C++;
- design and implement a class library as an abstraction, using the facilities of the C++ language and environment;
- implement efficient exception handling mechanisms in user-defined classes;
- Analyse and design and test C++ programs using OO methodology and a systematic approach.

Overall Assignment Instructions

For your assignment you are required to design, develop and test a specified object-oriented application using C++. The required application is described in the InterBanking Pty case-study outlined below.

- Throughout the semester you will be given clear instructions as to what is required to be implemented and submitted for review. Each of such submitted components will form part of your overall banking customer and account management system.
- Also, each submission will be marked and added to form your Unit assignment mark, which will total to 40% of your Unit total mark. The weighting of each submission will be indicated in the staged submission instructions.
- Whilst, you are allowed to discuss aspects of the assignment with your colleagues; all submitted material for marking should be unique to each student.
- The indicative marking scheme is shown below.

Submission instructions

The assignment submission should be made via the Unit's Blackboard assignments site. Penalties will be applied for late submission.

- All submission must include a Word or pdf document and all the source C++ code. The document should include your flowcharts and screenshots of the program execution.

- Your submission must be a single zipped file and the file name should be in the format of <yoursurname_studentnumber.zip>. For example, smith_1001234.zip.
- During the last laboratory session you will be required to demonstrate your system. Hence, the program must compile without errors and run in the ECU lab.
- Your submissions must be uploaded to the blackboard before or on the due date unless you have a medical certificate. Extensions must be applied in writing before the due date as per University policy.
- Also ensure that you keep a backup copy of all documentation and program files.

Case-Study Assignment: *InterBanking Pty*

A new Internet bank called InterBanking Pty commissioned you to develop their next generation customer and account management system. Through this case-study assignment you will be guided through the design, coding and testing of the required application.

At each stage of the project you will be given instructions as to what you require to implement and submit for review.

The task of this assignment is to produce 9 small programs (or components) their iterative integration to build the required customer and account management system.

Further specifications will be released as and when required, and will be clearly marked.

Getting Started

Stage 1:

1. You are required to produce a model of the customer account creation component of your system. For this you can use any modelling notation you are familiar with including ERD, OOD, etc.
2. You need to implement a C++ console-based application that performs the following functionality including:

    - Create an application that enables registered/authorised users to log in and create a new customer account and **BankAccount**. Use four-digit account numbers, but the program does not have to enforce this rule.
    - Develop a **struct** so that it contains public data fields that hold the **int** account number and **double** account balance and customer name and password.
    - Write a **main()** function that declares a **BankAccount** and allows you to enter data values for the **BankAccount** fields.
    - Echo the input – displays on screen the input details.
    - Use control constructs studied in lecture to develop a logging in part of the system, which will prompt users for their account numbers and name to enable them access to the program functionality such as viewing their bank balance.

At this stage do not worry about password validation, or data persistency (ie storing data for future usages). This will be developed next.

Stage 2:

Using the same structure created in **stage 1**, write a new (or amended) **main()** function, which you create two or more **BankAccount** objects for a given customer. Start with only two bank accounts per user. Add required statements to the **main()** function to do the following:

- Prompt the user for account numbers and beginning balances for his/her BankAccounts.
- For data entry validation, the bank account number should be between 1000 and 9999; issue an error message if the user does not enter a valid account number. The balance should not be negative; issue an error message if it is. Continue with the program only if the account numbers and balances are valid for both.
- Display the full account numbers and starting balances for the two accounts.
- Prompt the user for a dollar amount to be transferred from the first account to the second account. Issue an error message if the two accounts have the same account number, and do not carry out the transfer of funds.
- Issue an error message if the requested transfer causes the first account to fall below $0.00, and do not make the transfer.
- Issue a warning message if the transfer causes the balance in the first account to drop below $10.00, but make the transfer.
- Issue a warning message if the transfer causes the balance in the second account to be greater than $100,000.00, which is the highest amount that is federally insured.
- Display the ending balances after the transfer amount has been deducted from the first account and added to the second.
- For now, when an error message is issued, the program should terminate gracefully. For warning messages, the program continues to execute.

In this section you should whenever possible demonstrate in your code the use of loops, array and string processing, etc.

Submission Requirements:

Save the file as **BankAccount_1_2.cpp**, and submit it together with the assignment reports for Stage 1 & 2 submission requirements.

Deadline: 18th April 2015.

Stage 3:

Next you should add a third field that holds an annual interest rate earned on the account. Adding statements to the **main()** function developed previously to do the following:

- Prompt the user for account numbers, beginning balances, and interest rates for the two **BankAccount**s. We assume here that the interest rate will be different for different type of bank account. For instance, the interest rate will be higher for saving account compared to current accounts.
- The usual data validation developed in Stage 2 should now include the validation of the interest rate. For instance, the interest rate is between 0.01% and 15.0%. Also, the balance of each account should not be negative or over $100,000.00.
- Prompt the user for an automatic deposit account per month (in dollars and cents) and an automatic withdrawal amount per month (also in dollars and cents).
- The display a table that forecasts the balance every month for the term of each account. Calculate each month's balance as follows:
  - Calculate interest earned on the starting balance; interest earned is 1/12 of the annual interest rate times the starting balance.
  - Add the monthly automatic deposit.
  - Subtract the monthly automatic withdrawal.
  - Display the year number, month number, month-starting balance, and month-ending balance.
- Write a C++ function to simply read and write from a text file. This will be used to store each customer's bank accounts details.
- At this stage you don't need to provide any support for file search function, or other.

As this stage you should be able to demonstrate good use of all aspects studied so far, including use of functions to modularise your code, arrays, string processing for instance for passwords validation, and loops and control structures for data validation.

Submission Requirements:

Save the file as **BankAccount_3.cpp**, and submit it together with the updated assignment reports for Stage 3 submission requirements.

Deadline: 25th April 2015

Stage 4:

Based on your program functionality and structure developed so far, you are now required to re-organise your code and generalise your program's functional behaviour by using C++ functions to achieve the following:

- Write an **enterAccountData()** function that declares a local **BankAccount** object and prompt the user for values for each data field. The function returns a data-filled **BankAccount** object to the calling function. The usual data entry and output validation defined so far should hold.
- A **computeInterest()** function that accepts a **BankAccount** argument. Within the function, prompt the user for the number of years the account will earn interest. The function displays the ending balance of the account each year for the number of years entered based on the interest rate attached to the **BankAccount**.
- A **displayAccount()** function that displays the details of any **BankAccount** object passed to it.
- A **dataPersistency()** function that provides writing to and reading from files to store a specified customer's bank accounts details should be added.
- A **main()** function that declares a **BankAccount** object and continues to get **BankAccount** values from the user until the user enters a **BankAccount** with value 0 for the account number.
- Once the user has entered 0 indicating they do not wish to create further **BankAccount** objects, display the **BankAccount** details and a list of future balances based on the term the user requests for all entered accounts.

As this stage you should be able to demonstrate good use of all aspects studied so far, including use of functions to modularise your code, arrays, string processing for instance for passwords validation, and loops and control structures for data validation.

Submission Requirements:

Save the file as **BankAccount_4.cpp**, and submit it together with the updated assignment reports for Stage 4 submission requirements.

Deadline: 2nd May 2015

Stage 5:

Next you will develop a class that contains the fields and functions that a **BanckAccount** needs. Create a class containing private fields that hold the account number and the account balance. Include a constant static field that holds the annual interest rate (3%) earned on accounts at InterBanking Bank. Create three public member functions for the class, as follows:

- An **enterAccountData()** function that prompts the user for values for the account number and balance. The account number should not be less 1000 or greater than 9999, and the account balance should be greater than $0.00 and less than $100,000.00. In other words, for this type of account your program will not allow users to violate these two rules, thus the program should continue to prompt the user until valid values are entered.
- A **computeInterest()** function that accepts an integer argument that represents the number of years the account will earn interest on. The function displays the account number, then displays the ending balance of the account for each year, based on the interest rate attached to the **BankAccount**.
- A **displayAccount()** function that displays the details of the **BankAccount**.
- Create a **main()** function that declares an array of 10 **BankAccount** objects. After the first **BankAccount** object is entered, ask the user if he or she wants to continue.
- Prompt the user for **BankAccount** values until the user wants to quit or enters 10 **BankAccount** objects, whichever comes first. For each **BankAccount** entered, display the **BankAccount** details. Then prompt the user for a term with a value between 1 and 40 years inclusive. Continue to prompt the user until a valid value is entered. Then pass the array of **BankAccount** objects, the count of valid objects in the array, and the desired term (in terms of years) to a function that calculates and displays for each of the accounts the forecasted interest that will be earned each year (upto the specified term) based on the annual interest rate given above.

As this stage you should be able to demonstrate a good use of all aspects studied so far, including use of classes, functions to modularise your code, arrays, string processing for instance for passwords validation, and loops and control structures for data input and output validation.

Submission Requirements:

Save and submit your source code as **BankAccount_5.zip**, and submit it together with the updated assignment reports for Stage 5 submission requirements.

Deadline: 9th May 2015

Stage 6:

Object composition

You are required to develop a transaction class, which will be used to manage for instance the safe completion of the *BankAccount* transfer function. The latter was used to transfer a user selected amount of money between two accounts. To this end, you should make the necessary amendments to your previously developed applications (BankAccount_5) including the transfer function to work with the new Transaction class and behaviour.

You should design the transaction class with the necessary member data and functions required by your design. In particular, in your design you should make use of object composition studied in class to implement a transaction object that uses two bank account objects.

Design hints and requirements:

- Produce an outline design of your transaction class including how and with what other classes/objects it will interact with. Note that for this part of the assignment you are required to make use of object composition, as described above. It is entirely up to your choice as to what modelling notation to use for this, be it, UML, other or none of the above. But you should produce a simple but clear diagram, which will guide your design and implementation of you new functionality.
- For guidance only, your design could follow the following simplified transaction behaviour, which we used as an example in Lecture 08 (Fig. 1). Whereby, the transaction takes as input two specified *account* objects (*accountA and accountB)* and a required *amount* to be transferred from *accountA* to *accountB.* The *transaction* object will initialise the start of a given transaction and only commit the account model's updates (i.e. changing the *account balance*) if and only if the transaction has completed successfully. You could use the idea of a two state flag (say a Boolean variable), that indicates the start, pending and successful finish of a given transaction. Also, the transaction should implement a rollback function to for instance undo a given account balance change if required.

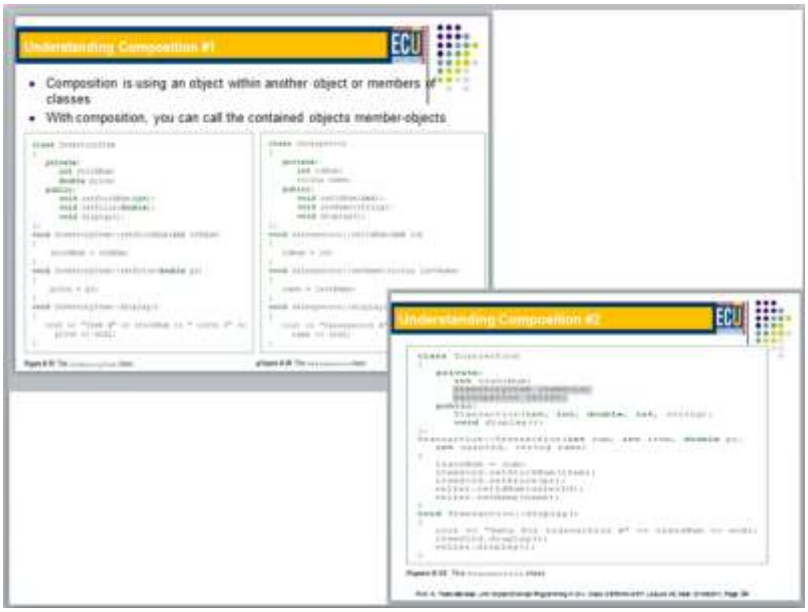

Figure 1: Transaction example use in Lecture 08 (pg 23-24).

As this stage you should be able to demonstrate a good use of all aspects studied so far, including use of classes, functions to modularise your code, arrays, string processing for instance for passwords validation, and loops and control structures for data input and output validation.

Submission Requirements:

In a single compressed file, submit your source code as **BankAccount_6.zip**, together with the usual document containing the design of the new behaviour (transaction) and its integration with the current version of your Bank_account program. Also include the relevant screenshots to illustrate the use and/or testing of your program.

Indicative Marking Schema

As a broad indication, the marking scheme will be 10% of this Stage mark allocated to the design (including the documentation of the new behaviour), 90% will be allocated to the correct (and error free) implementation of your design including the integration the overall program of the new transaction behaviour.

Deadline: 16th May 2015

Stage 7:

Inheritance, function templates and exception handling

- Create another class called ***BankBranch*** which contains the following fields
    - ***int*** BSB number,
    - ***String*** address,
    - ***int*** postcode.
    - Add a non-default constructor. This constructor accepts 3 arguments and sets each of the arguments to its respective data field. Prompt the user to specify the BSB number, address and postcode at the start of the session.
- In order to inherit from ***BankAccount*** class, in this section you are required to change the class definition as follows:
    - In previous steps, you declared the interest rate to be static and constant; remove those modifiers as well as the definition of the interest rate as 0.03. Also in the ***BankAccount*** class, add the member-object ***BankBranch*** homeBranch, which will contain the location details of the branch the bank account was created in.
    - Add a constructor to the class that accepts arguments for the account number, balance, interest rate, and the bank's branch details. The account number is automatically generated by your system, and should not be defined by the user as was required in previous versions of your system. Valid bank account numbers should be between 1000 and 9999. Also, both the balance and interest rate default to 0 if no argument is supplied to the constructor.
    - Modify the ***display*** function, if necessary, so that all ***BankAccount*** fields are displayed.
    - Remove the annual rate field and all references to it so that a ***BankAccount*** contains only an account number and a balance. Also remove the ***computeInterest()*** function if it is still part of your class.
    - Make any other necessary amendments to suit your design of the ***BankAccount*** class not mentioned above.
- Create two subclasses of ***BankAccount*** class called: ***SavingsAccount*** and ***CheckingAccount***, which are defined as follows:
    - ***SavingsAccount*** class has an interest rate field that is initialised to 3% when a ***SavingsAccount*** is constructed. Create an overloaded ***display*** member function, which displays all of the data for a ***SavingsAccount***. Also, add a constructor for the ***SavingsAccount*** class that requires a double argument that is used to set the interest rate.
    - ***CheckingAccount*** class has two new fields including a monthly fee and the number of checks allowed per month; both are entered from prompts within the constructor. Create an overloaded ***display*** member function to display all the data for a ***CheckingAccount***.

- Create a class named ***CheckingAccountWithInterest*** that descends from both ***SavingsAccount*** and ***CheckingAccount***.
    - This class provides for special interest-bearing checking accounts. When you create a ***CheckingAccountWithInterest*** object, force the interest rate to 0.02%, but continue to prompt for values for all the other fields.
    - Create an array of five ***CheckingAccountWithInterest*** accounts, prompt the user for values for each, and display the accounts.
- Finally after completing the development of the ***BankAccount*** class and its subclasses you are now required to add a report generator component as follows. This is provided to guide in the design of the template function, hence you can change the design of your template function to suit your existing design and code:
    - Create a template function named ***produceReport()*** that accepts two arguments as follows:
        - A string value for the report title
        - An array of accounts a given customer holds with your bank.
    - The ***produceReport()*** function displays/print to screen the details of each account you have passed to it in the array of accounts.
        - For presentation purposes, the report should
        - Start with the name of the report, and end with an "End of Report" message.
        - In the body of the report, use either a set of blank lines or a dashed line after each account details displayed on screen.
        - Each account details should be displayed in a tabular form. Thus, your template function will know how many columns each account type requires – ie depending on the number of attributes each class has.
    - Amend your ***main()*** function to reflect the changes specified above including: the addition of the ***produceReport()*** function, and showing at least one example of using exception handling.

Submission Requirements:

Your program should work as a standalone console application, and should provide all the functionalities developed so far including:

- password based access control,
- account generation, money transfer, report generation and bank account detail display.
- user interaction with appropriate input and output validation,
- etc.

In a single compressed file submit:

- your source code as **BankAccount_final.zip**,

- report containing the revised (or final) design of your application including your class diagram. Also include screenshots to illustrate the use and/or testing of your program.

Indicative Marking Schema

- As a broad indication, the marking scheme will be 10% of this Stage mark allocated to the design (including the documentation of the new behaviour), 90% will be allocated to the correct (and error free) implementation of your design including the integration the overall program of the new transaction behaviour.

Deadline:

- In class demonstration:
    - **30th May 2015**
    - Note: for off-campus students, the demonstration can be done either via students an avi recording of the application running, or Skype with date and time to be agreed with the lecturer.
- Final Assignment submission:
    - **28th May 2015**