# CS162 Project

## Overview

This is a GROUP PROJECT. Groups will be assigned the 1st week in February. Please look for an announcement about groups and how to interact in an online environment at that time.

For this course, you will build a more advanced version of a bank.  We used Account classes and the "idea" of a bank when we were introduced to functions and classes.  At this stage in our development, we know enough to create useful Account classes (and subclasses) that approximate systems used in production systems.

SameDiff Bank will be our basis for using more advanced parts of C++:

* Phase 1: Defining and designing a system using your brain and Unified Modeling Language (UML)
* Phase 2: Identifying considerations around data management and storage
* Phase 3: Using polymorphism to enhance code reuse and implementing this design in UML
* Phase 4: Create a simplified version of your design in C++, using the concepts of polymorphism, overloading, and comparable functions.

In the course, the project is broken up into smaller parts. With a solid understanding of Phase 1, 2, and 3, building the program is not that difficult.  Some students make the mistake of providing minimal input on Phases 1-3 and find that building the program is nearly impossible.  Think twice (or three times) and code once because a solid understanding of a problem usually makes the solution to the problem fairly simple.

Once you have completed Chapter 11, you are ready to begin Phase 1 of the project.

Please let me know if you have questions or concerns as we move through the process.

NOTE: This program will be used in classes following this one.

Steve Smiley  
Computer Science - CBC  
[ssmiley@columbiabasin.edu](mailto:ssmiley@columbiabasin.edu)

## Group 7:

Jordan Bassett (me)

Tayler Caufield

Austin Tesch

# Phase 1 Assignment

Scenario:

**Samediff Bank** has a 25-year-old banking account system. It was created using procedural programming. Samediff Bank needs to improve the security and maintainability of the system using an object-oriented programming (OOP) approach. Their bank manager has decided to hire you to develop, implement, and test a new OOP application using efficient data structures and programming techniques.

Phase 1 Requirements: Answers to the discussion questions, using Phase 1 Grading Rubric to optimize your grade.

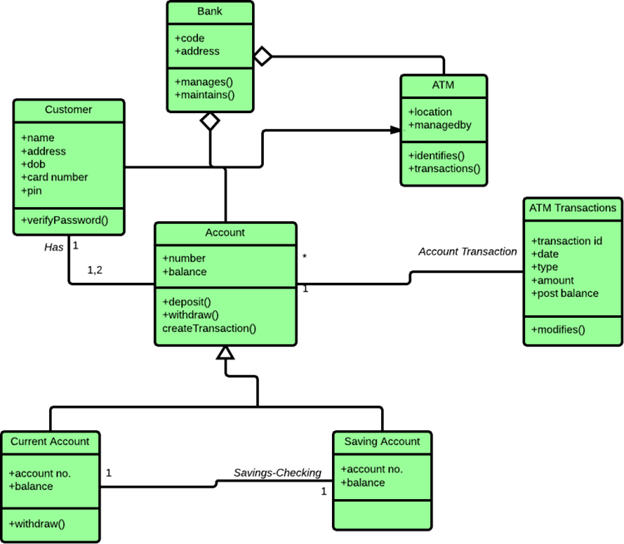
Use the following questions to help guide your team to accomplish tasks required in Phase 1:

1. (4 points possible) What are appropriate primitive data types needed for a bank account system? List them.
2. (2 points possible) When using a class for the scenario provided, will there be any static member variables? If yes, explain reasons to support your idea.
3. (2 points possible) What kind of constructors would you need? List them.

According to an account system, customers can have more than one type of an account.

1. (4 points possible) How many classes should you design?
2. (4 points possible ) Write the purpose of each class.
3. (2 points possible) What is the class relationship “has a relationship (composition)” , “is a relationship (inheritance)” or both?
4. (4 points possible) Jot down ideas what each class looks like. Then draw a UML diagram for all classes using correct symbols and class relationship.

The following diagram is a good example for “is a” and also a “has a” relationship.



A saving is an account.

An account has a bank.

See explanations below the diagram about symbols used.

Symbols used and meanings:

* The top part contains a class name.
* The second part contains data members.
* The last part contains member functions.

The minus symbol indicates a private member. The plus symbol indicates a public member. The # symbol if used, indicates a protected member.

Software for creating UML diagrams

You can use Powerpoint or Word to create UML diagrams, but it is a terrible idea and extremely time intensive to create the UML forms and fill them in.  Formatting is a nightmare.  A better option is to use software specifically designed to create UML diagrams.  A couple of options:

* Downloadable: Dia Diagramming - [http://dia-installer.de/Links to an external site.](http://dia-installer.de/) (Free and small)
* Online: LucidChart - [https://www.lucidchart.com/pages/Links to an external site.](https://www.lucidchart.com/pages/) (Free for a few diagrams - enough for our needs)
* Online: Creately - [https://try.creately.com/Links to an external site.](https://try.creately.com/) (Free, but ad-heavy)
* Online: Miro - [https://miro.com/Links to an external site.](https://miro.com/) (Free, but you have to answer an extensive questionnaire to access the tool)

Microsoft Visio is a part of the Microsoft Office 365 Suite; however, it is not included in the CBC suite of applications.  Do NOT buy Visio for this class; use on of the alternative provided above.

Jot down ideas about functions needed for each of the classes. Modify the UML diagram.

## Phase 1 Group Answers

1. **Appropriate primitive data types needed for a bank account system**
2. **Account Holder Name: string.**

Probably two using two string variables for first\_name and last\_name to keep names structured.

1. **Account Balance: double**

Provides the necessary precision to handle financial data accurately.

1. **Account Type: string**

Allows flexibility to store different account types as text (e.g., "Checking", "Savings", “Credit”).

1. **Interest Rate: double**

Precision is important for interest rate calculations, especially when compounding.

1. **Transaction Amount: double**

Precision is needed for accurate transaction handling.

1. **Transaction Date: time\_t**

Typically, a time\_t, often represented as a long int.

1. **Customer ID: int**

A unique integer identifier for each customer.

1. **PIN/Password: string**

Although stored as a string, this value should be hashed or encrypted for security.

1. **Static member variables**
   1. **The bank name:** Since all accounts belong to the same bank, the bank name can be defined as a static member variable. This allows all instances of the class to share the same bank name without duplicating data in each account object. Additionally, if the bank name ever needs to be updated, it can be changed in one place for all accounts.
   2. **Interest Rate for Savings Accounts**: All savings accounts share the same interest rate. This makes it easy to update the rate across all accounts if needed.
   3. **Transaction Fee**: a static member could store this value to apply consistently across accounts.
   4. **Total Number of Accounts**: Useful for generating unique account numbers or for tracking growth.
2. **Constructors needed**
3. **Default constructor**

A default constructor is useful for creating an empty account with default values. This could be helpful when initializing an account object before setting its specific details.

1. **Overloaded Constructor**

An overloaded constructor would allow the creation of an account with specific details provided at the time of creation. This constructor could take parameters like the account number, initial balance, account type, and customer ID to fully initialize an account when it is created.

1. **Seven Classes to be designed:** BankAccount, SavingsAccount, CheckingAccount. CreditAccount, Transaction, Customer, BankAccountContainer
2. **Purpose of each class:**
   * 1. **BankAccount (Base Class)**

Base class for different account types, including attributes and methods common to all accounts, like account\_number, balance, and basic deposit/withdraw methods.

* + 1. **SavingsAccount (Derived from BankAccount)**

Adds features specific to savings accounts, such as an interest\_rate attribute and methods for interest calculation. Inherits core functionality from BankAccount.

* + 1. **CheckingAccount (Derived from BankAccount)**

Adds checking-specific attributes and methods, such as an overdraft\_limit and potentially check-writing functions. Also inherits common functionality from BankAccount.

* + 1. **CreditAccount (Derived from BankAccount)**

Represents credit card accounts with features such as a credit\_limit, APR (annual percentage rate), and possibly reward points. This class can handle credit-specific methods, such as calculating monthly payments or interest on outstanding balances.

* + 1. **Transaction**

Manages the processing of various transaction types—deposits, withdrawals, and transfers. It could also store transaction-specific details, like transaction\_id, transaction\_date, transaction\_amount, and transaction\_type. As well as having methods to reverse or cancel any transactions.

* + 1. **Customer**

Stores customer details, such as name, address, contact\_info, and a list of associated BankAccount objects. This class could also include methods for managing and accessing the customer’s accounts as well as updating contact info.

* + 1. **BankAccount Container**

Acts as a manager for all BankAccount objects within the system. This container would oversee the creation, storage, and retrieval of accounts, allowing the system to efficiently manage multiple accounts across multiple customers. It could include methods to find accounts by account\_number or customer\_id and to handle bulk transactions if needed.

1. **Class relationships**
   * 1. **“Is-a” Relationship (Inheritance)**

BankAccount serves as the base class, and specific account types inherit from it. This defines an “is-a” relationship because SavingsAccount, CheckingAccount, and CreditAccount are all a BankAccount.

* + 1. “**Has-a” Relationship (Composition)**

Customer has a “has-a” relationship with BankAccount because each Customer can hold one or more BankAccount objects. This is composition because a Customer “has-a” BankAccount. A Transaction “has-a” date, amount, and is associated with one or more BankAccount objects (the accounts involved in the transaction).

The BankAccount Container also has a composition relationship with BankAccount objects, as it manages a collection of accounts within the system.

A diagram of a bank account

Description automatically generated

# Phase 2 Assignment

Samediff bank would like to use your data structures. They would like you to be able to save a list of accounts and load a list of accounts. Implement saving and loading of multiple occurrences of your data structures.

Phase 2 Requirements: Answers to the discussion questions, using Phase 2 Grading Rubric to optimize your grade.

Use the following questions to help guide your team to accomplish tasks required in Phase 2:

1. (2 points possible) What are advantages using a binary file?
2. (2 points possible) What are advantages using a text file?
3. (2 points possible) If there is a very large amount of data, what file should you use, text file or binary file?
4. (2 points possible) If you are to use a binary file, what should be a data type for a customer’s name?
5. (2 points possible) If you use a string object and want to convert it to a c-string, what is a function used to accomplish the task? Provide a short code sample.
6. (2 points possible) If you have multiple objects consider that a customer may have multiple accounts. Can you write objects to a file? Provide a reference section in Chapter 13.
7. (2 points possible) If you want to group related variables needed as a record for a bank checking account as a record, develop that data structure.
8. (2 points) What input validation should you implement when reading information from a file from disk? Provide a code sample with an error message(s).

### Phase 2 Rubric

| **Criteria** | **Ratings** |
| --- | --- |
| Question 1- Binary file and advantages | 2 pts More than one advantage is clearly stated.  1 pts One advantage is clearly stated.  0 pts No answer provided. |
| Question 2- Text file and advantages | 2 pts More than one advantage is clearly stated.  1 pts One advantage is clearly stated.  0 pts No answer provided. |
| Question 3- File type to hold large data | 2 pts Best decision provided.  1 pts Not the best decision provided.  0 pts No answer provided. |
| Question 4- Data type for a binary file name | 2 pts Best decision provided.  1 pts Not the best decision provided.  0 pts No answer provided. |
| Question 5- string object conversion to a c-string | 2 pts A correct function selected. Also, there a code sample provided.  1 pts A correct function selected. But there is no code sample provided.  0 pts No answer provided. |
| Question 6- Write an object to a file (?) | 2 pts The answer is correct answer and there is a correct reference provided to support the answer.  1 pts The answer is correct answer but there is no correct reference provided to support the answer.  0 pts No answer provided. |
| Question 7- Record creation | 2 pts Use a correct data structure to group related data.  1 pts Use an incorrect data structure to group related data  0 pts No answer provided. |
| Question 8- Validation | 2 pts Use a correct input validation, and a code sample provided.  1 pts Use a correct input validation, but no code sample provided.  0 pts No answer provided. |

## Phase 2 Group Answers

1. **(2 points possible) What are advantages using a binary file?**

One advantage of using a binary file is that the code can execute faster since a computer already operates at the binary level. By keeping it in a binary file you don't have to run as many conversions. Binary files maintain the exact memory representation of data, which helps in keeping complex data types and structures intact. Text files require encoding each character (e.g., "12345" takes at least 5 bytes in a text file, but only 4 bytes as a binary integer). This overhead can make text files larger when storing numbers, floating-point data, or complex structures.

Binary files are also smaller than text files, especially with large amounts of data. Text files require encoding each character (e.g., "12345" takes at least 5 bytes in a text file, but only 4 bytes as a binary integer). This overhead can make text files larger when storing numbers, floating-point data, or complex structures.

As binary files store data in a raw format, it makes them impossible to read or interpret directly without specialized software or code. While not a replacement for proper encryption, binary storage makes it harder for casual attempts to read or tamper with the data. This is important for handling banking data in the project.

**2. (2 points possible) What are advantages using a text file?**

An advantage of using a text file is that they are easier to read and modify. It is also easier to detect an error in the text file rather than in a binary file. Also, text files can be read on any system, whereas binary file formats may vary between systems due to differences in byte order. This makes text files more portable with cross-platform compatibility.

1. **(2 points possible) If there is a very large amount of data, what file should you use, text file or binary file?**

For large amounts of data you should use a binary file since they can store data more compactly. Binary files are also faster to read and write.

1. **(2 points possible) If you are to use a binary file, what should be a data type for a customer’s name?**

The data type for a customer’s name should be std::string. They have a dynamic size, can be concatenated and searched easily, and have a large number of useful functions in the string library.

1. **(2 points possible) If you use a string object and want to convert it to a c-string, what is a function used to accomplish the task? Provide a short code sample.**

Use name.c\_str() to convert a string object to a c-string:

|  |
| --- |
| #include <iostream> #include <string> int main() {  std::string name = "First Last";  const char\* c\_name = name.c\_str(); *//Converts string to const char\**  std::cout << "C-style string: " << c\_name << std::endl;  return 0; } |

1. **(2 points possible) If you have multiple objects consider that a customer may have multiple accounts. Can you write objects to a file? Provide a reference section in Chapter 13.**

Yes, you can write multiple objects to a file using file streams. If we have a customer who can have multiple accounts, we can create classes for both Customer and Account and write these objects to a file in binary or text format. The process of transforming complex networks of objects interconnected through pointers into a form that can be stored in files is called object serialization (Chapter 13.8, p2344).

1. **(2 points possible) If you want to group related variables needed as a record for a bank checking account as a record, develop that data structure.**

|  |
| --- |
| struct CheckingAccount {  int accountNumber; *// Account number of the checking account*  std::string accountHolder; *// Name of the account holder*  double balance; *// Current balance of the account*  std::string accountType; *// Type of the account (e.g., "Checking")*  std::string dateOpened; *// Date when the account was opened* }; |

1. **(2 points) What input validation should you implement when reading information from a file from disk? Provide a code sample with an error message(s).**

Check if the file exists and if your program is able to access it. Validate that the file content is in the expected format. Ensure that the data types of the content match what your program expects.

|  |
| --- |
| *// example check if file was opened successfully*  if (!inputFile.is\_open()) {  std::cerr << "Error: Could not open the file." << std::endl;  return 1;  }  *// example validate the data format*  if (!(lineStream >> value)) {  std::cerr << "Error: Invalid data format on line " << lineNumber << ". Expected an integer." << std::endl;  continue;  }  *// example validate that the value is non-negative* if (value < 0) {  std::cerr << "Error: Value on line " << lineNumber << " should not be negative." << std::endl;  continue;  } |

# Phase 3 Assignment

Samediff banks manager is excited to consider updating their account system. An expert has advised that they would be able to increase both security and ease of maintenance by using object-oriented concepts such as polymorphism. Create an inheritance hierarchy that a bank will use to represent customers’ bank accounts. All customers of Samediff bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. Each month they process interest earned and penalties on each account. Specific types of accounts exist. Each type of account you will need to consider in your inheritance hierarchy is listed below.

Steps taken for this phase:

1. (12 points possible) Revise your UML diagram to contain a customer or person class, account class, checking class, savings class, moneymarket class, cd class. Any functions that deal with the balance of an account must be virtual in a base class that derived classes override. Such as checking and account classes.
2. (3 points possible) Connect the classes using an inheritance hierarchy and object composition (if any) for Samediff Bank’s account system.
3. (2 points possible) Based on the information below, what functions should be polymorphic and use dynamic binding for security?
4. (2 points possible) How do you keep track of multiple accounts from a single customer? Explain your thoughts....

|  |  |  |  |
| --- | --- | --- | --- |
| Account Type | Monthly Interest (APY) | Withdrawal Penalty | Maturity Penalty |
| Savings | 1.05% | Balance < $1000.00=$50.00 | $0.00 |
| Checking | 0.02 | $0.00 | $0.00 |
| Certificate of Deposit (CD) | 3 months 2.5%  6 months 3%  1 year 5% | <3 months  deducts 10%  <6 months deducts 20%  <12 months deducts 50% | |
| Money Market (MMDA) | 1.25% | Balance < $10,000 = $200 | $0.00 |

### Phase 3 Rubric

| **Criteria** | **Ratings** |
| --- | --- |
| **UML** | 12 pts - The diagram contains all of information needed. Also, all of the symbols used are correct.  6 pts - The diagram contains most of information needed. Also, most of the symbols used are correct.  1 pts - The diagram does not contain all information needed. Also, some symbols used are correct.  0 pts - No answer provided. |
| **Class Relationships** | 3 pts - All class relationships are appropriately used and symbolized.  2 pts - Most of class relationships are appropriately used and some symbols used are correct.  1 pts - Most of class relationships are inappropriately used and symbolized.  0 pts - No Marks |
| **Question -Polymorphic/Virtual Functions** | 2 pts - Functions are appropriately identified as virtual functions.  1 pts - Functions are inappropriately identified as virtual functions.  0 pts - No Marks |
| **Question - Multiple accounts and management** | 2 pts - Demonstrate ideas based on theories.  1 pts - Demonstrate ideas without using theories.  0 pts - No answer provided. |

## Phase 3 Group Answers

**1 & 2) Changes and Updates to the UML (see last page & attachment) include:**

Added CertificateOfDepositBase as a parent class with subclasses (OneYearCD, ThreeMonthCD, SixMonthCD) to handle specific penalties, durations, and APY values.

Added Penalty for penalty-specific calculations. Added BalanceHandler to manage balance-related logic.

Marked withdraw() and deposit() in BankAccount as **[virtual]** to make it polymorphic.

Created MoneyMarketBase and its derived MMDA

Clarified that BankAccountContainer now centrally manages accounts, simplifying the Customer class.

**3) Based on the information below, what functions should be polymorphic and use dynamic binding for security?**

withdraw(): Some account types handle penalties differently: SavingsAccount applies penalties for low balances, CDAccount applies maturity penalties. These penalties need to be specific to the account type, so this function should be polymorphic.

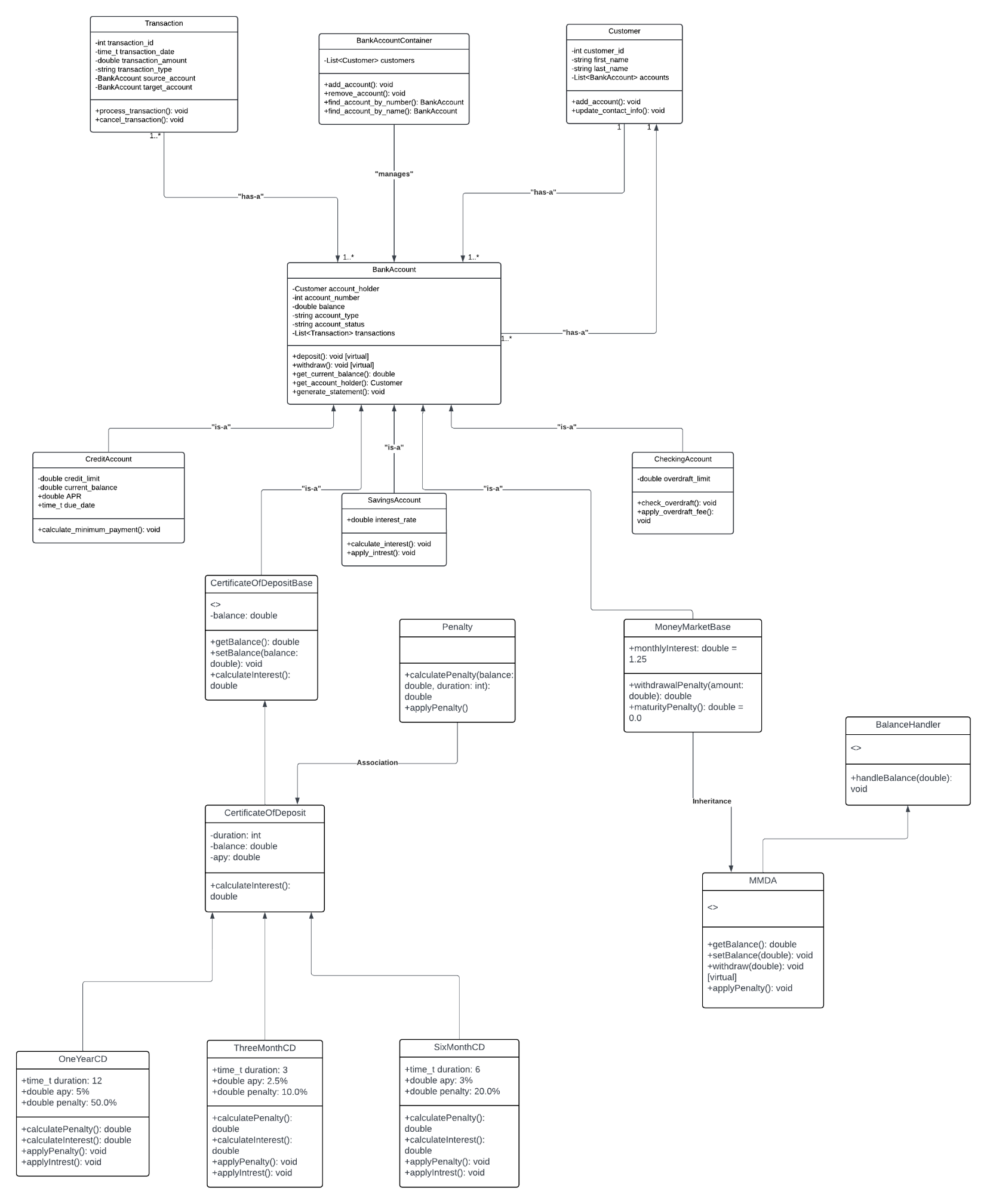
deposit(): For most accounts, deposit logic might be the same. However, CreditAccount might have special rules (e.g., paying off credit or applying a minimum payment). Making this function virtual allows for those customization.

applyMonthlyInterest(): Accounts like SavingsAccount and MoneyMarketAccount earn monthly interest. Since the interest rate and calculations are specific to each type of account, this function should be polymorphic.

generateStatement(): For reporting of different account details (including penalties, interest, or overdraft limits), this function can also be virtual.

**4) How do you keep track of multiple accounts from a single customer? Explain your thoughts....**

To keep track of multiple accounts from a single customer, allocate arrays for each customer using std::vector with the Customer class. That allows pulling up the whole array and all the accounts or use a pointer to only pick one a single account from the list. For storage, write customer and account data using a unique customer ID. Accounts can be serialized into binary files for loading when the program starts so that data is not lost between sessions.



# Phase 4 Assignment

Implement your account system so that you can load and save all data to and from a binary file. The data should be loaded into an STL vector. You need to include a menu system to add customers and accounts, delete customers and accounts and print accounts sorted by balance as well as print customers sorted by last name.

### Phase 4 Rubric

| Criteria | Ratings |
| --- | --- |
| **Unified Modeling Language (UML)** | 4 pts - UML diagram illustrates all classes with appropriate class relationships. (“has a” and/or “is a”)  3 pts - UML diagram illustrates most classes with appropriate class relationships. (“has a” and/or “is a”)  2 pts - UML diagram illustrates few classes with appropriate class relationships. (“has a” and/or “is a”)  1 pts - UML diagram illustrates multiple classes that have no relationships. (“has a” and/or “is a”)  0 pts - No illustration of a UML diagram. |
| **Method Reusability (Extract Method)** | 4 pts - A class defines accessor and mutator methods for each private data member. A mutator method manipulates a data member. An accessor returns a data member.  3 pts - A class defines accessor and mutator methods for most private data members. A mutator method manipulates a data member. An accessor returns a data member.  2 pts - A class defines accessor and mutator methods for a few private data members. A mutator method manipulates a data member. An accessor returns a data member.  1 pts - A class defines accessor and mutator methods for at least one private data member. A mutator method manipulates a data member. An accessor returns a data member.  0 pts - A class defines accessor and mutator methods for at least one private data member. A mutator method manipulates a data member. An accessor returns a data member.  0 pts - No accessor and mutator methods are defined. |
| **Class Reusability (Extract Class)** | 4 pts - If using inheritance, “is a” relationship, the super/base class must contain attributes that are needed for all derived classes. Each derived class must contain attributes only specific to itself. If using composition, “has a” relationship, a larger class must contain a component object of another class. The component class must contain all attributes that are needed in the larger class. For example, a class Person, has an object of a class Job.  3 pts - If using inheritance, “is a” relationship, the super/base class must contain attributes that are needed for most derived classes. Each derived class must contain attributes only specific to itself. If using composition, “has a” relationship, a larger class must contain a component object of another class. The component class contains most attributes that are needed in the larger class.  2 pts - If using inheritance, “is a” relationship, the super/base class must contain attributes that are needed for a few derived classes. Each derived class must contain attributes only specific to itself. If using composition, “has a” relationship, a larger class must contain a component object of another class. The component class contains a few attributes that are needed in the larger class.  1 pts - If using inheritance, “is a” relationship, the super/base class must contain attributes that are needed for a few derived classes. Not all derived classes contain attributes only specific to themselves. If using composition, “has a” relationship, a larger class must contain a component object of another class. The component class contains at least one attribute that is needed in the larger class.  0 pts - No evidence of class reusability. |
| **Access specification** | 4 pts - A class has data members defined as private. If using inheritance, a class defines a protected data member appropriately. A class defines public methods that are called via the class objects.  3 pts - A class does not define most data members as private. If using inheritance, a class defines a protected data member appropriately. A class defines public methods that are called via the class objects.  2 pts - A class defines a few data members as private. If using inheritance, a class defines a protected data member appropriately. A class defines public methods that are called via the class objects.  1 pts - A class defines a few data members as private. If using inheritance, a class defines a protected data member inappropriately. A class defines public methods that are not called via the class objects.  0 pts - A class does not define data members as private. |
| **Class Methods/Functions (Move Methods)** | 4 pts - Class methods generate all of the required tasks  3 pts - Class methods generate most of the required tasks  2 pts - Class methods generate more than 50% of the required tasks.  1 pts - Class methods generate less than 50% of the required tasks.  0 pts - Class methods do not generate any required tasks. |
| **Default Constructor** | 4 pts - Each class contains the definition of the default constructor with private data members initialized, using default values.  3 pts - Each class contains the definition of the default constructor with private data members initialized, not using default values.  2 pts - Each class contains the definition of the default constructor without private data members initialized.  1 pts - Each class contains the definition of a constructor.  0 pts - No constructor defined. |
| **Object Instantiation in the main()** | 4 pts - Main contains all object instantiation and user input interaction. Arguments are passed to backend utility objects, methods and constructors.  3 pts - Main contains most object instantiation and user input interaction. Most arguments are passed to backend utility objects, methods and constructors.  2 pts - Main contains few object instantiation and user input interaction. A few arguments are passed to backend utility objects, methods and constructors  1 pts - Main contains few object instantiation and user input interaction. A few arguments are passed to backend utility objects, methods and constructors  0 pts - Main does not contain objects. |
| **Screenshots of outputs** | 4 pts - Strong evidence working programs that test and meet all required tasks.  3 pts - Strong evidence working programs that test and meet a few of the required tasks.  3 pts - Strong evidence working programs that test and meet most of the required tasks.  2 pts - Strong evidence working programs that test and meet a few of the required tasks.  1 pts - Strong evidence working programs that test and meet at least one of the required tasks.  0 pts - No evidence of working program outputs. |
| **Documentation** | 4 pts - Sufficient comments that explain all methods and variables.  3 pts - Sufficient comments that explain most methods and variables.  2 pts - Sufficient comments that explain a few methods and variables.  1 pts - Sufficient comments that explain at least one method and variable.  0 pts - No evidence of documentation. |

**File Organization**

1. **Header Files**:
   * **BankAccount.h**: Base class and derived account classes.
   * **Customer.h**: Customer class declaration.
   * **BankAccountContainer.h**: BankAccountContainer class declaration.
   * **Common.h**: Common includes or utility functions.
2. **Source Files**:
   * **BankAccount.cpp**: Implementation of BankAccount and its derived classes.
   * **Customer.cpp**: Implementation of the Customer class.
   * **BankAccountContainer.cpp**: Implementation of the BankAccountContainer class.
   * **main.cpp**: Main entry point.