**CSC122**

**Group Project#2**

**Due 07/20/2025 by 11:59 pm**

**250 Points**

# ***Exercise #1:***

Write C++ program to maintain employee information for a business. There are two groups of employees, “manager” and “worker”. The suggested approach to complete the program is as follows:

1. Declare a class (using the name “employee”) with the following data items

Employee Number: a 9-digit integer number

Name: up to 30 characters

Age: a 2-digit integer number

Gender Code: single character

Education: two-character code. For example HS(High School),

AD(Associate Degree), CD(Coll. Degree), MD(Masters Degree)

1. A worker class, which **includes** the above data items and has the following additional data items:

Hiring Date: in MM/DD/YYYY format

Dept. Name: 4 characters, for example ACCT(Accounting), SALE(Sales)

HUMN(Human Resources)

Hourly Rate: float

1. A manager class which **includes** the data items in class employee and has these additional data items:

Date of Promotion: in MM/DD/YYYY format

Department Name: 4 characters

Job Title: 20 characters

Yearly Salary: float

Your program should be menu-driven, such that when it starts, the main menu will display options such as: add manager, add worker, display manager, display worker, modify manger, modify worker,…..etc. By selecting an option, corresponding function of different classes will be executed.

Limit the number of managers to 3 and workers to 5 such that if a user attempts to add 4th manager a warning should be displayed informing the limit has been reached. Same with adding 6th worker.

# ***Exercise #2:***

Define a pure abstract base class called **BasicShape**. The **BasicShape** class should have the following members:

Private Member Variable:

**area**: A double used to hold the shape’s area.

Public Member Functions:

**getArea**: This function should return the value in the member variable area.

**calcArea**: This function should be a pure virtual function.

Next, define a class named **Circle**. It should be derived from the **BasicShape** class. It should have the following members:

Private Member Variables:

**centerX**: a long integer used to hold the x coordinate of the circle’s center

**centerY**: a long integer used to hold the y coordinate of the circle’s center

**radius**: a double used to hold the circle’s radius

Public Member Functions:

**constructor**: accepts values for centerX, centerY, and radius. Should call the overridden calcArea function described below

**getCenterX**: returns the value in centerX

**getCenterY**: returns the value in centerY

**calcArea**: calculates the area of the circle (area = 3.14159 \* radius \* radius) & stores the result in the inherited

member area.

Next, define a class named **Rectangle**. It should be derived from the **BasicShape** class. It should have the following members:

Private Member Variables:

**width**: a long integer used to hold the width of the rectangle

**length**: a long integer used to hold the length of the rectangle

Public Member Functions:

**constructor**: accepts values for width and length. Should call the overridden calcArea function described below.

**getWidth**: returns the value in width.

**getLength**: returns the value in length.

**calcArea**: calculates the area of the rectangle (area = length \* width) and stores the result in the inherited member

area.

After you have created these classes, create a driver program that defines a Circle object and a Rectangle object. Demonstrate that each object properly calculates and reports its area.

# ***Exercise #3:***

Banks offer various types of accounts, such as savings, checking, certificates of deposit, and money market, to attract customers as well as meet their specific needs. Two of the most commonly used accounts are savings and checking. Each of these accounts has various options. For example, you may have a savings account that requires no minimum balance but has a lower interest rate. Similarly, you may have a checking account that limits the number of checks you may write. Another type of account that is used to save money for the long term is certificate of deposit (CD).

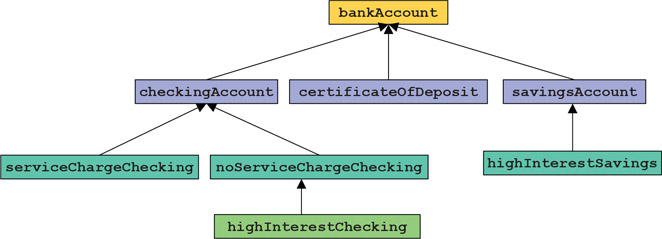
In this exercise, you use abstract classes and pure virtual functions to design classes to manipulate various types of accounts. For simplicity, assume that the bank offers three types of accounts: savings, checking, and certificate of deposit, as described next.

**Savings accounts**: Suppose that the bank offers two types of savings accounts: one that has no minimum balance and a lower interest rate and another that requires a minimum balance and has a higher interest rate.

**Checking accounts**: Suppose that the bank offers three types of checking accounts: one with a monthly service charge, limited check writing, no minimum balance, and no interest; another with no monthly service charge, a minimum balance requirement, unlimited check writing, and lower interest; and a third with no monthly service charge, a higher minimum requirement, a higher interest rate, and unlimited check writing.

**Certificate of deposit (CD**): In an account of this type, money is left for some time, and these accounts draw higher interest rates than savings or checking accounts. Suppose that you purchase a CD for six months. Then we say that the CD will mature in six months. The penalty for early withdrawal is stiff. Figure below shows the inheritance hierarchy of these bank accounts.

Figure: Inheritance hierarchy of banking accounts



Note that the classes bankAccount and checkingAccount are abstract. That is, we cannot instantiate objects of these classes. The other classes in Figure are not abstract.

**bankAccount**: Every bank account has an account number, the name of the owner, and a balance. Therefore, instance variables such as **name**, **accountNumber**, and **balance** should be declared in the abstract class **bankAccount**. Some operations common to all types of accounts are retrieve account owner’s name, account number, and account balance; make deposits; withdraw money; and create monthly statements. So, include functions to implement these operations. Some of these functions will be pure virtual.

**checkingAccount**: A checking account is a bank account. Therefore, it inherits all the properties of a bank account. Because one of the objectives of a checking account is to be able to write checks, include the pure virtual function **writeCheck** to write a check.

**serviceChargeChecking**: A service charge checking account is a checking account. Therefore, it inherits all the properties of a checking account. For simplicity, assume that this type of account does not pay any interest, allows the account holder to write a limited number of checks each month, and does not require any minimum balance. Include appropriate named constants, instance variables, and functions in this class.

**noServiceChargeChecking**: A checking account with no monthly service charge is a checking account. Therefore, it inherits all the properties of a checking account. Furthermore, this type of account pays interest, allows the account holder to write checks, and requires a minimum balance.

**highInterestChecking**: A checking account with high interest is a checking account with no monthly service charge. Therefore, it inherits all the properties of a no service charge checking account. Furthermore, this type of account pays higher interest and requires a higher minimum balance than the no service charge checking account.

**savingsAccount**: A savings account is a bank account. Therefore, it inherits all the properties of a bank account. Furthermore, a savings account also pays interest.

**highInterestSavings**: A high-interest savings account is a savings account. Therefore, it inherits all the properties of a savings account. It also requires a minimum balance.

**certificateOfDeposit**: A certificate of deposit account is a bank account. Therefore, it inherits all the properties of a bank account. In addition, it has instance variables to store the number of CD maturity months, interest rate, and the current CD month.

Write the definitions of the classes described in this programming exercise and a program to test your classes.

# ***Exercise #4:***

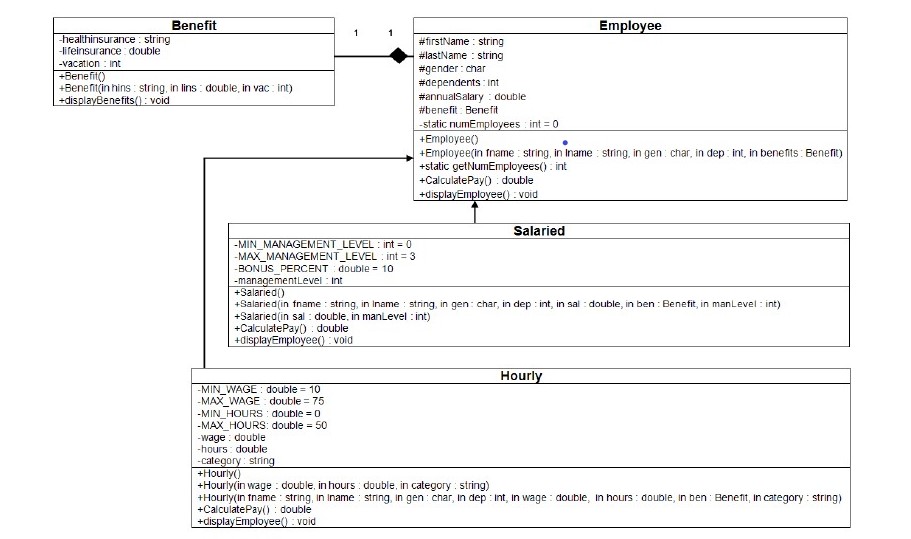
Using the following UML diagram as a reference, create the classes. It is common practice to leave out the accessors and mutators (getters and setters) from UML class diagrams, since there can be so many of them. Unless otherwise specified, it is *assumed* that there is an accessor (getter) and a mutator (setter) for every class attribute.

**Understanding the UML Class Diagram**

1. The first section specifies the class name.
2. The second section specifies the attributes (Data Members)
3. The third section specifies the behaviors (Methods)

The first character specifies the access specifier value, where

* "-" means that the class member is private
* “#” means that the class member is protected
* "+" means that the class member is public.

****

**Code the Benefit Class**

An employee typically has benefits, so:

1. Create a Benefits class.

**Creating the Employee Class**

1. The default constructor should set the attributes as follows: firstName = "not given", lastName = "not given", gender = "U" (for unknown), dependents = 0, and annualSalary = 20,000.
2. The multi-arg constructor should initialize all of the attributes using values passed in using its parameter list.
3. As shown in the Class diagram, each attribute should have a "getter" to retrieve the stored attribute value, and a "setter" that modifies the value.
4. The calculatePay( ) method of the Employee class should return the value of annual salary divided by 52 (return annualSalary / 52;).
5. The displayEmployee() method should display all the attributes of the Employee object in a well-formatted string with logical labels applied to each attribute. Don't forget to call calculatePay from within the displayEmployee method in order to display the Employee's weekly pay as well!
6. Integrate the Benefit class into the Employee class. (Composition)

**Creating the Salaried Class**

1. Using the above UML Diagram, create the Salaried class, ensuring to specify that the Salary class inherits from the Employee class.
2. For each of the constructors listed in the Salaried class, ensure to invoke the appropriate base class constructor and pass the correct arguments to the base class constructor. This will initialize the protected attributes and update the numEmployees counter.
3. The valid management levels are 0, 1, 2, and 3, and should be implemented as a constant.
4. Override the calculatePay method to add a 10 percent bonus for each of the management levels (i.e., bonus percentage = managementLevel \* .10). The bonus percentage should be implemented as a constant.
5. Override the displayEmployee() method to add the management level to the employee information.

**Creating the Hourly Class**

1. Using the UML Diagram, create the Hourly classes, ensuring to specify that the Hourly class inherits from the Employee class.
2. For each of the constructors listed in the Hourly class, ensure to invoke the appropriate base class constructor and pass the correct arguments to the base class constructor. This will initialize the protected attributes and update the numEmployees counter.
3. The valid category types are "temporary", "part time", and "full time".
4. The provided hours must be more than 0 hours and less than 50 hours, and the limits should be implemented as constants.
5. The provided wage must be between 10 and 75, and the limits should be implemented as constants.
6. Override the calculatePay method by multiplying the wages by the number of hours.
7. Override the Employee setAnnualSalary method and set the annual salary by multiplying the weekly pay by 50.
8. Override the displayEmployee() method to add the category to the hourly employee information.

**Testing (main)**

1. Create at least one object of Employee, Hourly, and Salaried employee.
2. For each object created, display the number of employees created.
3. For each object created, write statements to exercise each of the public methods listed in the Class diagram.
4. For each object created, invoke the object's displayEmployee() method to display the employee's information.

**Compile and Test**

When done, compile and run your code. Then, debug any errors until your code is error-free. Check your output to ensure that you have the desired output, modify your code as necessary, and rebuild.

# ***Exercise #5:***

**(CarbonFootprint Abstract Class: Polymorphism)**

Using an abstract class with only pure virtual functions, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming.

Create three small classes **unrelated** by inheritance—classes Building, Car and Bicycle. Give each class some unique appropriate attributes and behaviors that it does not have in common with other classes. Write an abstract class CarbonFootprint with only a pure virtual getCarbonFootprint method. Have each of your classes inherit from that abstract class and implement the getCarbonFootprint method to calculate an appropriate carbon footprint for that class (check out a few websites that explain how to calculate carbon footprints).

Write an application that creates objects of each of the three classes, places pointers to those objects in a vector of CarbonFootprint pointers, then iterates through the vector, polymorphically invoking each object’s getCarbonFootprint method. For each object, print some identifying information and the object’s carbon footprint.