CS3201: OBJECT ORIENTED PROGRAMMING LABORATORY

Topic: Virtual function, runtime polymorphism, compile time polymorphism

Lab: 06

Date: April 02, 2025

EXECUTION TASKS

1. Design a hierarchical payroll management system using abstract classes and pure virtual functions. Implement a base class **Employee** with attributes such as *name, ID, and baseSalary*, along with a pure virtual function *calculateSalary()* to enforce salary computation in derived classes. Create specialized employee types including **Manager**, who receives a fixed base salary, a performance bonus, and company shares linked to profit; **Developer**, who earns a base salary, overtime pay, and an additional bonus for technical certifications; **SalesExecutive**, whose compensation includes a base salary plus a commission percentage of revenue generated; and **Intern**, who receives a stipend along with a project completion bonus. Store employee objects using pointers and leverage polymorphism to dynamically compute salaries. Implement **operator overloading** for == and < to compare employee salaries and enable sorting in descending order based on earnings. Extend the system to allow dynamic role transitions, such as an Intern being promoted to Developer, while ensuring the original object reference remains intact.

2. Design a custom string formatting utility similar to **std::format in C++**, utilizing function overloading and operator overloading to dynamically format different data types. Implement a class **StringFormatter** with overloaded methods such as *format(int value)*, which converts an integer to a string representation; *format(double value, int precision)*, which formats a floating-point number with a specified decimal precision; *format(bool value)*, which converts a boolean to either "true" or "false"; and *format(string value, int width, char fillChar)*, which formats a string by applying padding using a specified width and fill character. Enhance the system by overloading the << *operator* to support chained formatting, enabling expressions like cout << sf << "Value: " << sf.format(123) << ", Pi: " << sf.format(3.14159, 2);. Additionally, overload the + *operator* to concatenate formatted strings seamlessly.

3. Design a data analysis engine using function overloading and operator overloading to process multiple data formats efficiently. Implement a class **DataAnalyzer** with overloaded methods such as *analyze(int[], int size)*, which computes statistical measures like mean, median, and mode for integer datasets; *analyze(double[], int size)*, which calculates variance and standard deviation for floating-point datasets; and *analyze(std::string data)*, which performs word frequency analysis on text data. Enhance the system by overloading the << *operator* to allow seamless reporting, enabling expressions like *cout << analyzer*; to print summary statistics. Additionally, overload the + *operator* to combine multiple datasets dynamically, allowing complex multi-source data analysis.

4. Design an banking system. The system should include a base class **BankAccount**, which should have two key attributes: *accountNumber* and *balance*. The base class should contain a pure virtual function *withdraw(double amount)* to enforce withdrawal policies, as well as a *deposit(double amount)* method to allow adding funds to the account. Additionally, include a virtual function *detectFraud(double amount)* in the base class, which will flag suspicious activities based on the withdrawal amount. Create a **SavingsAccount** that restricts withdrawals if the balance falls below a minimum threshold, and applies a dynamic monthly interest rate based on the balance; a **CurrentAccount** that allows overdraft up to a specified limit but imposes penalties when the overdraft limit is exceeded; and a **CryptoAccount** that applies withdrawal fees based on real-time volatility in simulated market conditions. Each of these derived classes should override the withdraw and detectFraud functions to implement their specific behaviors.To track all deposit and withdrawal transactions, create a *mixin* class **TransactionHistory** that logs transaction details. This class should use multiple inheritance to be mixed into the different account types. Each account type should have the ability to display its transaction history, as well as a mechanism to detect fraud if a withdrawal exceeds a specific threshold percentage of the account balance.