# COMS-280 Final Project Phase 4

1. Interface for interest-bearing accounts

Class InterestBearing {

Public:

Virtual void applyInterest() = 0;

Virtual ~InterestBearing() = default;

};

* 1. Let SavingAccount inherit from both

Class SavingsAccount : public BankAccount, public InterestBearing {

}; // implements applyInterest()

* 1. Allows runtime polymorphism based on capabilities, not just type

If (auto ib = dynamic\_cast<InterestBearing\*>(account)) {

Ib->applyInterest();

}

1. Factory pattern for account creation – makes it easier to add new account types:

Class AccountFactory {

Public:

Static unique\_ptr<BanlAccount> createAccount(const string& type, const string& name, doub

If (type == “savings”) return make\_unique<SavingsAccount>(name, balance, extra);

If (type == “checking”) return make\_unique<CheckingAccount>>(name, balance, extra);

Throw invalid\_argument(“Unknown account type”);

}

};

1. Generic Lambda for Transaction Display

Auto displayTransactions = [] (const BankAccount& acc) {

Acc.displayTransactionHistory();

};

1. Lambda-based Filters for AccountList

Auto filterAccounts = [](CustmerList& list, double minBalance) {

For (auto\* node = list.begin(); node != nullptr; node = node->NEXT) {

IF (node->account->getBalance() > minBalance)

Node->account->display();

}

};

### **Documentation and Explanation**

Rationale Behind Template Implementation and Exception Handling

The banking system was designed to simulate real-world financial account operations using a modular and maintainable object-oriented approach. Templates and exception handling were incorporated to improve the flexibility, reusability, and robustness of the codebase.

* Templates were used to support type-generic operations—most notably in the calculatePercentage<T> function, which allows interest and percentage-based financial calculations to be performed on any numeric data type.
* Exception Handling ensures the system remains stable when encountering invalid operations (e.g., overdrafts or withdrawals beyond available balance). This provides user-friendly feedback instead of allowing the system to crash.

### Challenges Encountered During Implementation

* Generic Type Conversion: Care had to be taken when handling template types in calculatePercentage<T> to avoid precision loss during division.
* Dynamic Type Handling: Implementing InterestBearing via a separate interface required runtime type-checking (dynamic\_cast) to determine whether an account supports interest.
* Memory Management: Use of raw pointers in the customer linked list required meticulous handling of memory deallocation to avoid leaks; unique\_ptr was used in combination with manual deletion in the list destructor.

### Benefits of Using Templates and Robust Exception Handling

1. Templates:

* Increased code reuse and reduced duplication.
* Improved maintainability as calculations are centralized in a single templated function.
* Enabled potential future extension for other numeric types (e.g., float, long double).

1. Exception Handling:

* Increased code reuse and reduced duplication.
* Improved maintainability as calculations are centralized in a single templated function.
* Enabled potential future extension for other numeric types (e.g., float, long double).

### Assumptions and Limitations

1. Assumptions:

* All monetary values are stored as double by default.
* Users interact via a simple console interface and provide valid input formats.
* Account names are unique for lookup purposes.

1. Limitations:

* No persistent storage (data is lost upon program termination).
* Only two types of accounts are implemented.
* No support for concurrency or multi-user access.
* The customer list is implemented as a singly linked list with linear search.

### **Reflection and Future Improvements**

### Implementation Reflection

The process of integrating templates and exception handling into this project has significantly deepened my understanding of how to build flexible, type-safe, and robust applications.

* Templates were intuitive to implement but required caution with type casting to avoid unintended behavior (especially during division).
* Exception Handling provided a clean mechanism to separate error-checking logic from regular business logic, enhancing code readability and structure.

What I Learned and Areas for Improvement

1. Lessons Learned:

* Templates can abstract repetitive logic and simplify financial calculations across different data types.
* Catching specific exceptions improves user feedback and simplifies debugging.
* Separation of concerns (e.g., utilities for overdraft and interest) improves modularity.

1. Areas for Improvement:

* Introduce custom exception classes for more granular error categorization (e.g., InsufficientFundsException, InvalidAccountTypeException).
* Replace the singly linked list with a std::map or another container for efficient account lookup.
* Validate all user inputs (e.g., non-numeric entries or negative values).

Proposed Enhancements

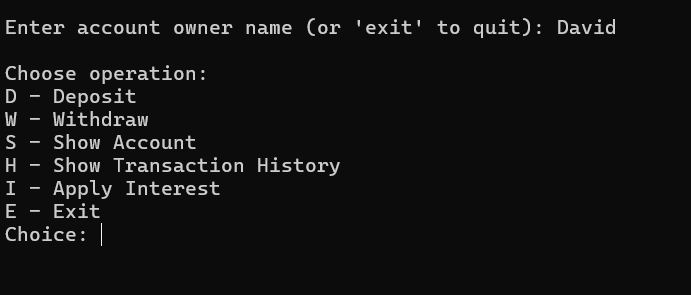
* Persistence - Save and load customer/account data using file I/O or database integration.
* GUI Interface - Replace the console interface with a user-friendly GUI (e.g., Qt or C++/CLI).
* Interest Rate Variability - Allow savings interest rates to vary over time or based on account activity.
* Unit Testing - Introduce automated tests using a framework like Google Test for regression testing.

Broader Implications

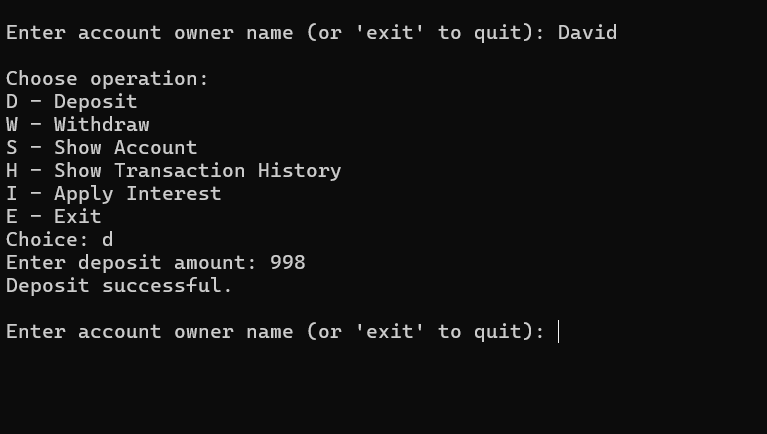
Templates and exception handling are essential in building large-scale, real-world systems, especially in domains like banking where precision and reliability are critical. The use of templates promotes maintainability and adaptability, while robust exception handling ensures systems can fail gracefully without compromising data integrity or user trust.

## **Testing**

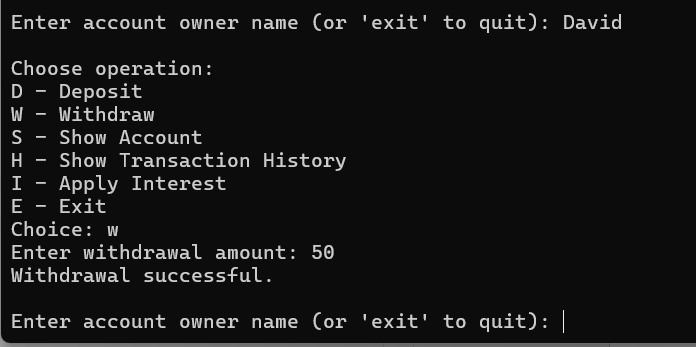
Enter account name



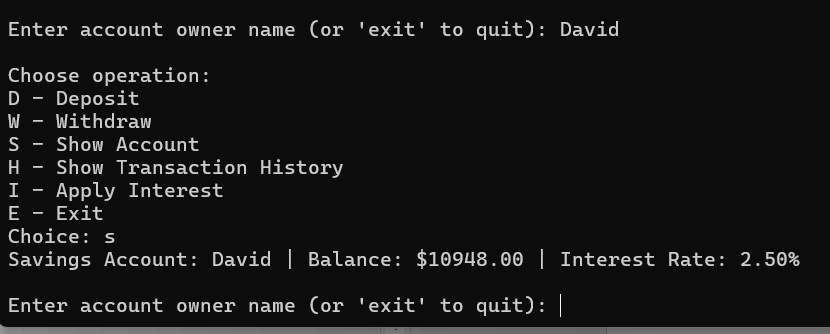
Choose Deposit



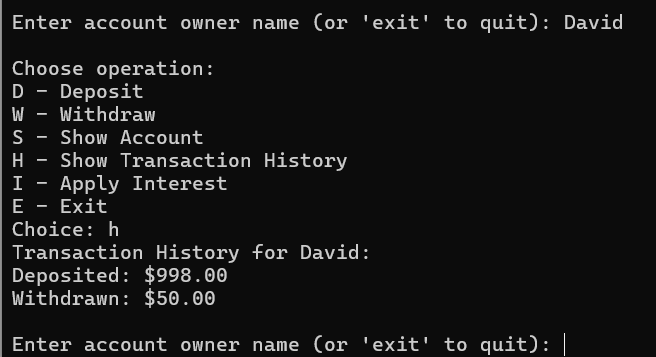
### Choose withdrawal



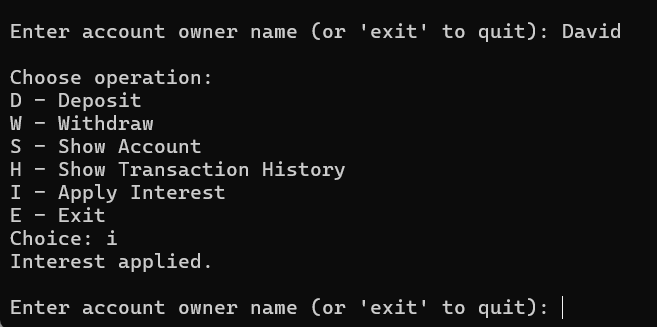
Choose show account



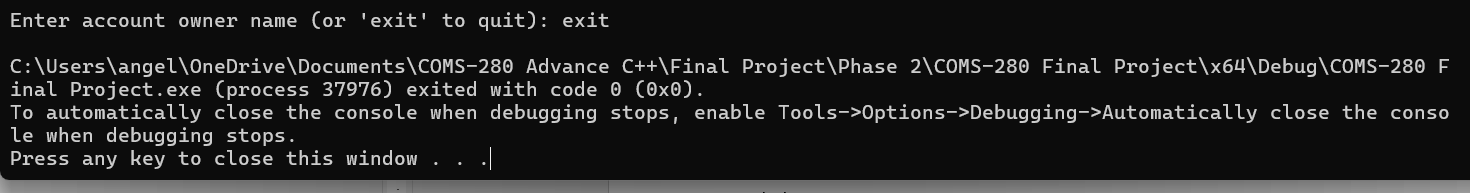
Choose show transaction history



Choose apply interest



Choose exit



#include <iostream>

#include <list>

#include <memory>

#include <iomanip>

#include <sstream>

#include <stdexcept>

#include <functional>

using namespace std;

/\*\*

\* Template function for calculating a percentage.

\* Supports generic numeric types for financial computations.

\*/

template <typename T>

T calculatePercentage(T value, T rate) {

return value \* (rate / static\_cast<T>(100));

}

/\*\*

\* Utility class to handle interest calculation for savings accounts.

\* Uses the generic percentage template.

\*/

class InterestCalculator {

public:

static double calculateInterest(double balance, double rate) {

return calculatePercentage(balance, rate);

}

};

/\*\*

\* Utility class to enforce overdraft rules in checking accounts.

\*/

class OverdraftProtection {

public:

static bool canWithdraw(double balance, double overdraft, double amount) {

return amount <= (balance + overdraft);

}

};

/\*\*

\* Interface for interest-bearing accounts.

\* Enables runtime polymorphism based on feature rather than type.

\*/

class InterestBearing {

public:

virtual void applyInterest() = 0;

virtual ~InterestBearing() = default;

};

/\*\*

\* Base class representing a generic bank account.

\* Implements polymorphic behavior through virtual functions.

\* Provides common interface for all account types.

\*/

class BankAccount {

protected:

string owner;

double balance;

list<string> transactionHistory;

// Helper method to format currency with two decimal places

string formatAmount(double amount) const {

ostringstream stream;

stream << fixed << setprecision(2) << amount;

return stream.str();

}

public:

BankAccount(string name, double initialBalance) : owner(name), balance(initialBalance) {}

virtual ~BankAccount() = default;

// Pure virtual methods to be implemented by derived classes

virtual void deposit(double amount) = 0;

virtual void withdraw(double amount) = 0;

virtual void display() const = 0;

void addTransaction(const string& transaction) {

transactionHistory.push\_back(transaction);

}

void displayTransactionHistory() const {

cout << "Transaction History for " << owner << ":\n";

for (const auto& transaction : transactionHistory) {

cout << transaction << endl;

}

}

double getBalance() const { return balance; }

string getOwner() const { return owner; }

};

/\*\*

\* Derived class representing a savings account.

\* Implements InterestBearing interface to support advanced polymorphism.

\*/

class SavingsAccount : public BankAccount, public InterestBearing {

private:

double interestRate;

public:

SavingsAccount(string name, double balance, double rate)

: BankAccount(name, balance), interestRate(rate) {

}

void deposit(double amount) override {

balance += amount;

addTransaction("Deposited: $" + formatAmount(amount));

}

void withdraw(double amount) override {

if (amount > balance)

throw runtime\_error("Insufficient funds");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

void applyInterest() override {

double interest = InterestCalculator::calculateInterest(balance, interestRate);

deposit(interest);

addTransaction("Interest Applied: $" + formatAmount(interest));

}

void display() const override {

cout << "Savings Account: " << owner << " | Balance: $" << fixed << setprecision(2) << balance

<< " | Interest Rate: " << interestRate << "%\n";

}

};

/\*\*

\* Derived class representing a checking account.

\* Uses OverdraftProtection utility class for safe withdrawal.

\*/

class CheckingAccount : public BankAccount {

private:

double overdraftLimit;

public:

CheckingAccount(string name, double balance, double overdraft)

: BankAccount(name, balance), overdraftLimit(overdraft) {

}

void deposit(double amount) override {

balance += amount;

addTransaction("Deposited: $" + formatAmount(amount));

}

void withdraw(double amount) override {

if (!OverdraftProtection::canWithdraw(balance, overdraftLimit, amount))

throw runtime\_error("Overdraft limit exceeded");

balance -= amount;

addTransaction("Withdrawn: $" + formatAmount(amount));

}

void display() const override {

cout << "Checking Account: " << owner << " | Balance: $" << fixed << setprecision(2) << balance

<< " | Overdraft Limit: $" << overdraftLimit << "\n";

}

};

/\*\*

\* Factory class for creating accounts using C++14 features.

\* Demonstrates use of make\_unique and simplified control flow.

\*/

class AccountFactory {

public:

static unique\_ptr<BankAccount> createAccount(const string& type, const string& name, double balance, double extra = 0.0) {

if (type == "savings") return make\_unique<SavingsAccount>(name, balance, extra);

if (type == "checking") return make\_unique<CheckingAccount>(name, balance, extra);

throw invalid\_argument("Unknown account type");

}

};

/\*\*

\* Struct representing a node in the customer linked list.

\*/

struct CustomerNode {

unique\_ptr<BankAccount> account;

CustomerNode\* next;

CustomerNode(unique\_ptr<BankAccount> acc)

: account(move(acc)), next(nullptr) {

}

};

/\*\*

\* Manages a singly linked list of bank accounts.

\*/

class CustomerList {

private:

CustomerNode\* head;

public:

CustomerList() : head(nullptr) {}

~CustomerList() {

while (head) {

auto\* temp = head;

head = head->next;

delete temp;

}

}

void addCustomer(unique\_ptr<BankAccount> account) {

auto\* newNode = new CustomerNode(move(account));

newNode->next = head;

head = newNode;

}

bool deleteCustomer(const string& name) {

auto\* curr = head;

CustomerNode\* prev = nullptr;

while (curr) {

if (curr->account->getOwner() == name) {

if (prev) prev->next = curr->next;

else head = curr->next;

delete curr;

return true;

}

prev = curr;

curr = curr->next;

}

return false;

}

BankAccount\* getCustomerByName(const string& name) {

for (auto\* curr = head; curr != nullptr; curr = curr->next) {

if (curr->account->getOwner() == name)

return curr->account.get();

}

return nullptr;

}

void displayAll() const {

for (auto\* curr = head; curr != nullptr; curr = curr->next) {

curr->account->display();

}

}

};

/\*\*

\* Interface for interacting with customer accounts using advanced polymorphism.

\*/

void performBankingOperations(CustomerList& customers) {

string name;

char choice;

double amount;

while (true) {

cout << "\nEnter account owner name (or 'exit' to quit): ";

cin >> name;

if (name == "exit") break;

auto\* account = customers.getCustomerByName(name);

if (!account) {

cout << "Account not found.\n";

continue;

}

cout << "\nChoose operation: \nD - Deposit\nW - Withdraw\nS - Show Account\nH - Show Transaction History\nI - Apply Interest\nE - Exit\nChoice: ";

cin >> choice;

switch (choice) {

case 'D': case 'd':

cout << "Enter deposit amount: ";

cin >> amount;

account->deposit(amount);

cout << "Deposit successful.\n";

break;

case 'W': case 'w':

cout << "Enter withdrawal amount: ";

cin >> amount;

try {

account->withdraw(amount);

cout << "Withdrawal successful.\n";

}

catch (const exception& e) {

cout << "Error: " << e.what() << endl;

}

break;

case 'S': case 's':

account->display();

break;

case 'H': case 'h': {

auto displayTransactions = [](const BankAccount& acc) {

acc.displayTransactionHistory();

};

displayTransactions(\*account);

break;

}

case 'I': case 'i': {

if (auto\* ib = dynamic\_cast<InterestBearing\*>(account)) {

ib->applyInterest();

cout << "Interest applied.\n";

}

else {

cout << "This account does not support interest calculation.\n";

}

break;

}

case 'E': case 'e':

return;

default:

cout << "Invalid choice.\n";

}

}

}

/\*\*

\* Entry point: Initializes customers using AccountFactory.

\*/

int main() {

CustomerList customers;

customers.addCustomer(AccountFactory::createAccount("savings", "Laurie", 5000, 2.5));

customers.addCustomer(AccountFactory::createAccount("checking", "Larry", 1000, 500));

customers.addCustomer(AccountFactory::createAccount("savings", "David", 10000, 2.5));

customers.addCustomer(AccountFactory::createAccount("checking", "Luis", 2000, 500));

performBankingOperations(customers);

return 0;

}