CCP6124 - OBJECT ORIENTED PROGRAMMING AND DATA STRUCTURES

TRIMESTER 2520

Assignment

TUTORIAL SECTION:
TT1L

GROUP 4

Name	Student ID
EBA MOHAMED ABBAS AHMED (Leader)	242UC243BE
LAMA M. R. SIAM	242UC243B4
ARINA AGHAEE	1221303277
MEREY ABILKHAN	241UT24016

Table of Contents

Executive Summary

This assignment demonstrates advanced object-oriented programming concepts through two comprehensive systems:

1. **Bank Account Management System** - Implements inheritance hierarchy, smart pointers, and advanced error handling using singly linked lists
2. **Warehouse Inventory System** - Custom template-based Stack and Queue implementations with real-world workflow modeling

Both systems feature professional-grade error handling, file persistence, and user-friendly interfaces that simulate real-world applications.

Question 1: Bank Account Management System Using Singly Linked List

1.1 System Overview and Design Rationale

The Bank Account Management System implements a comprehensive banking solution using object-oriented principles and dynamic memory management. The design choices were made to simulate real-world banking operations while demonstrating advanced C++ concepts.

Key Design Decisions:

- **Singly Linked List**: Chosen for dynamic memory allocation and O(1) insertion at head
- **Smart Pointers**: Used (unique_ptr) for automatic memory management and exception safety
- **Inheritance Hierarchy**: Implemented to support different account types with specialized behaviors

1.2 Core Architecture

1.2.1 Class Hierarchy Design

Account Class Hierarchy:

Account (Base Class)
├── SavingsAccount (Interest-bearing accounts)
└── CheckingAccount (Overdraft protection)

Design Rationale: The inheritance structure allows polymorphic behavior while maintaining code reusability. Virtual functions enable runtime dispatch for account-specific operations.

Base Account Class Implementation:

```
class Account {
protected:
    string accountNumber, customerName;
    double balance;
public:
    Account(const string& accNum = "", const string& custName = "", double initialBalance
= 0.0)
        : accountNumber(accNum), customerName(custName), balance(initialBalance) {}

    virtual ~Account() = default;
```

```
    // Getters
    string getAccountNumber() const { return accountNumber; }
    string getCustomerName() const { return customerName; }
    double getBalance() const { return balance; }

    // Setters
    void setBalance(double newBalance) { balance = newBalance; }

    virtual void displayDetails() const {
        cout << "Account Type: " << getAccountType() << "\nAccount Number: " <<
accountNumber
             << "\nCustomer Name: " << customerName << "\nBalance: $" << balance <<
endl;
    }

    virtual string getAccountType() const { return "Basic Account"; }
    virtual bool canWithdraw(double amount) const { return amount <= balance; }
    virtual double getAvailableBalance() const { return balance; }
    virtual void showSpecialFeatures() const {
        cout << "No special features for basic account." << endl;
    }
};
```

Why Protected Members: Account details are protected to allow derived class access while maintaining encapsulation from external classes.

1.2.2 Hybrid Memory Management Architecture

Smart Pointer Node Structure:

```
struct Node {
    unique_ptr<Account> account; // Smart pointer for Account objects
    Node* next;                  // Raw pointer for list navigation
    Node(unique_ptr<Account> acc) : account(std::move(acc)), next(nullptr) {}
};
```

Why This Hybrid Approach:

- **Smart Pointers for Objects:** unique_ptr<Account> ensures automatic cleanup of Account objects and prevents memory leaks
- **Raw Pointers for Structure:** Using raw pointers for next maintains simple list traversal and avoids circular dependency issues with smart pointers
- **Move Semantics:** std::move(acc) transfers ownership efficiently without copying
- **Professional Design:** You don't need smart pointers for every pointer - only for resource management

This is actually a sophisticated approach used in professional codebases where smart pointers manage object lifetimes while raw pointers handle structural relationships.

1.2.3 Advanced Error Handling System

Three-Tier Validation Architecture:

1. **Format Validation:** Ensures correct data types and formats
2. **Business Logic Validation:** Checks business rules (account existence, sufficient funds)
3. **Recovery Mechanisms:** Allows users to retry operations without menu navigation

Core Error Recovery Functions:

```
// Function to clear input buffer
void clearInputBuffer() {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}

// Function to ask for retry with specific error messages
bool askForRetry(const string& errorType) {
    string choice;
    cout << "Error in " << errorType << ". Do you want to retry? (y/n): ";
    getline(cin, choice);
    return (choice == "y" || choice == "Y");
}
```

Enhanced Menu System with Error Handling:

```
// menu selection with error handling
int getMenuChoice() {
    int choice;
    while (true) {
```

```
        cout << "\n===== Bank Account Management System =====\n"
             << "1. Add account\n2. Display all accounts\n3. Search by account number\n"
             << "4. Deposit\n5. Withdraw\n6. Delete account\n7. Show account info\n8.
Exit\n";
        cout << "Enter choice: ";

        if (!(cin >> choice) || choice < 1 || choice > 8) {
            cout << "Error: Invalid input. Please enter a number between 1 and 8." << endl;
            clearInputBuffer();
            if (!askForRetry("menu selection")) {
                return -1; // User chose not to retry
            }
            continue;
        }
        cin.ignore(); // Clear the newline after numeric input
        return choice;
    }
}
```

Why This Approach: Traditional systems force users back to main menu on errors. Our approach provides immediate correction opportunities, improving user experience significantly.

1.2.4 Input Validation Functions

Account Number Validation with Business Logic:

```
// account number input with validation
string getAccountNumber(BankSystem& bankSystem, bool shouldExist = true) {
    string accNum;
    while (true) {
        cout << "Enter account number (numbers only, min 3 digits): ";
        getline(cin, accNum);

        // Basic validation
        if (accNum.empty() || accNum.length() < 3) {
            cout << "Error: Account number must be at least 3 characters long." << endl;
            if (!askForRetry("account number input")) {
                return "";
            }
            continue;
        }

        // Check if contains only digits
        bool validFormat = all_of(accNum.begin(), accNum.end(), [](char c) { return isdigit(c);
});
        if (!validFormat) {
```

```
            cout << "Error: Account number can only contain numbers." << endl;
            if (!askForRetry("account number input")) {
                return "";
            }
            continue;
        }

        // Check existence
        Account* existingAcc = bankSystem.searchByAccountNumber(accNum);
        if (shouldExist && !existingAcc) {
            cout << "Error: Account number " << accNum << " not found." << endl;
            if (!askForRetry("account number input")) {
                return "";
            }
            continue;
        }

        if (!shouldExist && existingAcc) {
            cout << "Error: Account number " << accNum << " already exists." << endl;
            if (!askForRetry("account number input")) {
                return "";
            }
            continue;
        }

        return accNum;
    }
}
```

Customer Name Validation:

```
// customer name input with comprehensive validation
string getCustomerName() {
    string custName;
    while (true) {
        cout << "Enter customer name: ";
        getline(cin, custName);

        if (custName.empty() || custName.length() < 2) {
            cout << "Error: Name must be at least 2 characters long." << endl;
            if (!askForRetry("account input process")) {
                return "";
            }
            continue;
        }

        bool validName = all_of(custName.begin(), custName.end(), [](char c) {
            return isalpha(c) || c == ' ';
        });
        bool hasLetter = any_of(custName.begin(), custName.end(), [](char c) {
```

```
        return isalpha(c);
    });

    if (!validName) {
        cout << "Error: Name can only contain letters and spaces." << endl;
        if (!askForRetry("account input process")) {
            return "";
        }
        continue;
    }

    if (!hasLetter) {
        cout << "Error: Name must contain at least one letter." << endl;
        if (!askForRetry("account input process")) {
            return "";
        }
        continue;
    }

    return custName;
  }
}
```

Amount Validation with Business Rules:

```
// amount input with validation
double getAmount() {
  double amount;
  while (true) {
    cout << "Enter amount: $";
    if (!(cin >> amount)) {
      cout << "Error: Invalid input. Please enter a valid number." << endl;
      clearInputBuffer();
      if (!askForRetry("amount input")) {
        return -1;
      }
      continue;
    }

    if (amount <= 0) {
      cout << "Error: Amount must be positive." << endl;
      clearInputBuffer(); // Clear the input buffer after detecting negative amount
      if (!askForRetry("amount input")) {
        return -1;
      }
      continue;
    }

    clearInputBuffer(); // Clear the input buffer before returning valid amount
    return amount;
  }
```

```
}
```

Features:

- **Existence Checking:** Validates whether account should or shouldn't exist
- **Duplicate Prevention:** Prevents creation of duplicate accounts
- **Format Validation:** Ensures only numeric characters, minimum length requirements
- **Name Validation:** Only accepts alphabetic characters and spaces, requires minimum 2 characters, prevents empty or invalid names, must contain at least one alphabetic character

1.2.5 File Persistence Implementation

Data Structure Preservation Strategy:

```
void saveAccountsToFile() {
    ofstream outFile(DATA_FILE);
    if (!outFile) {
        cout << "Error: Could not save to file." << endl;
        return;
    }

    int count = 0;
    for (Node* current = head; current; current = current->next) {
        Account* acc = current->account.get();
        outFile << "TYPE:" << acc->getAccountType() << "\nNUMBER:" <<
acc->getAccountNumber()
                << "\nNAME:" << acc->getCustomerName() << "\nBALANCE:" <<
acc->getBalance() << "\n";

        if (auto* savings = dynamic_cast<SavingsAccount*>(acc))
            outFile << "INTEREST_RATE:" << savings->getInterestRate() << "\n";
        else if (auto* checking = dynamic_cast<CheckingAccount*>(acc))
            outFile << "OVERDRAFT_LIMIT:" << checking->getOverdraftLimit() << "\n";

        outFile << "----------\n";
        count++;
    }
    outFile.close();
    cout << count << " accounts saved." << endl;
}
```

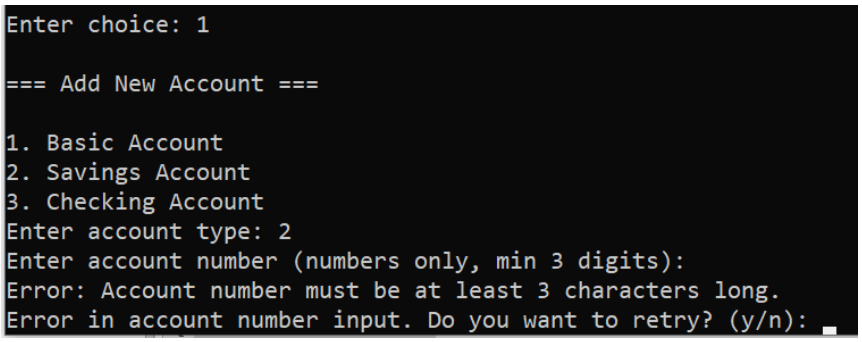
Why This Format: The tagged format allows reliable parsing and supports different account types without data loss. The separator ensures clear record boundaries.

1.3 Error Handling Examples with Screenshots

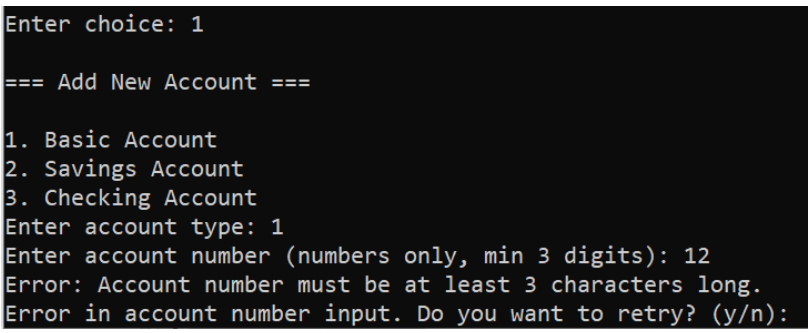
The system provides comprehensive error messages for various validation scenarios:

Input Validation Error Messages:

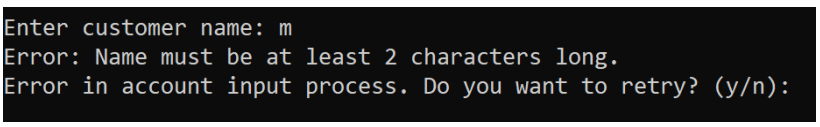
Error message for input cannot be empty:



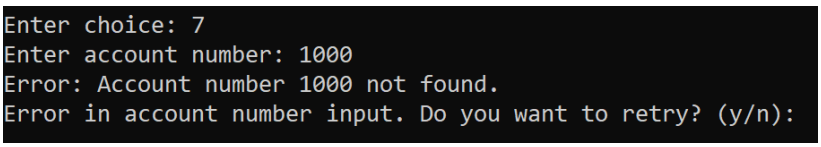
Error message for account number too short:



Error message for name too short:



Error message for Account number not found:



Error message for Account number already exists:

```
Enter choice: 1

=== Add New Account ===

1. Basic Account
2. Savings Account
3. Checking Account
Enter account type: 3
Enter account number (numbers only, min 3 digits): 1002
Error: Account number 1002 already exists.
Error in account number input. Do you want to retry? (y/n):
```

Error message for invalid inputs:

```
Enter initial balance: $@$100
Error: Invalid input. Please enter a valid number.
Error in account input process. Do you want to retry? (y/n):
```

Error message for negative inputs:

```
Enter initial balance: $-20
Error: Balance cannot be negative.
Error in account input process. Do you want to retry? (y/n):
```

Error message when entering letters in account number:

```
Enter choice: 1

=== Add New Account ===

1. Basic Account
2. Savings Account
3. Checking Account
Enter account type: 1
Enter account number (numbers only, min 3 digits): tyu
Error: Account number can only contain numbers.
Error in account number input. Do you want to retry? (y/n):
```

Error message when entering numbers in customer name:

```
Enter customer name: 123
Error: Name can only contain letters and spaces.
Error in account input process. Do you want to retry? (y/n):
```

Transaction Security:

- ❖ Positive amount validation for all transactions
- ❖ Balance verification before withdrawals
- ❖ Overdraft limit enforcement for checking accounts
- ❖ Clear transaction confirmations with before/after balances

Error message for duplicate account number:

```
Enter account number (numbers only, min 3 digits): 1001
Error: Account number 1001 already exists.
Error in account number input. Do you want to retry? (y/n):
```

Name Validation:

- Only accepts alphabetic characters and spaces
- Requires minimum 2 characters
- Prevents empty or invalid names
- Must contain at least one alphabetic character

Error message for negative deposit amount:

```
Enter choice: 4
Enter account number (numbers only, min 3 digits): 1001
Enter amount: $-20
Error: Amount must be positive.
Error in amount input. Do you want to retry? (y/n): y
Enter amount: $
```

Error message for insufficient funds with available balance shown:

```
Enter choice: 5
Enter account number (numbers only, min 3 digits): 1002
Enter amount: $400
Error: Insufficient funds! Available: $100
Error in withdrawal process. Do you want to retry? (y/n): y
Enter account number (numbers only, min 3 digits): 300
Error: Account number 300 not found.
Error in account number input. Do you want to retry? (y/n): n
```

1.4 Detailed Input Validation Functions

Initial Balance Input:

```
// initial balance input with comprehensive validation
double getInitialBalance() {
    double balance;
    while (true) {
        cout << "Enter initial balance: $";
        if (!(cin >> balance)) {
            cout << "Error: Invalid input. Please enter a valid number." << endl;
            clearInputBuffer();
```

```
            if (!askForRetry("account input process")) {
                return -1;
            }
            continue;
        }

        if (balance < 0) {
            cout << "Error: Balance cannot be negative." << endl;
            clearInputBuffer(); // Clear the input buffer after detecting negative balance
            if (!askForRetry("account input process")) {
                return -1;
            }
            continue;
        }

        clearInputBuffer(); // Clear the input buffer before returning valid balance
        return balance;
    }
}
```

1.5 Polymorphic Operations Implementation

Runtime Polymorphism in Action:

```
Account* acc = bankSystem.searchByAccountNumber(accNum);
acc->displayDetails();      // Calls appropriate derived class method
acc->showSpecialFeatures(); // Runtime polymorphism in action
```

Runtime Behavior:

- **SavingsAccount** shows interest calculations
- **CheckingAccount** displays overdraft limits
- **Basic Account** shows standard features

1.6 Technical Implementation Details

1. Memory Management

- Uses unique_ptr for automatic Account object management
- Custom destructor properly cleans up linked list
- Move semantics for efficient object transfers
- Raw pointers for simple list navigation

2. Error Handling:

- Comprehensive input validation with immediate retry

- Exception handling for file operations
- User-friendly error messages with specific guidance

3. Data Persistence

- Automatic save/load functionality
- Structured file format for reliable data recovery
- Polymorphic serialization handles different account types correctly

4. Account Creation Process:

```
// account creation process with comprehensive validation
bool createNewAccount(BankSystem& bankSystem) {
    while (true) {
        cout << "\n=== Add New Account ===" << endl;

        // Get account type
        int type = getAccountType();
        if (type == -1) return false; // User chose not to retry

        // Get account number
        string accNum = getAccountNumber(bankSystem, false); // false = should not exist
        if (accNum.empty()) return false; // User chose not to retry

        // Get customer name
        string custName = getCustomerName();
        if (custName.empty()) return false; // User chose not to retry

        // Get initial balance
        double balance = getInitialBalance();
        if (balance == -1) return false; // User chose not to retry

        // Create and add account
        auto acc = bankSystem.createAccount(type, accNum, custName, balance);
        if (acc && bankSystem.addAccount(std::move(acc))) {
            cout << "Account added successfully!" << endl;
            return true;
        } else {
            cout << "Error: Failed to create account." << endl;
            if (!askForRetry("account input process")) {
                return false;
            }
            // Continue the loop to retry the entire process
        }
    }
}
```

5. Transaction Processing:

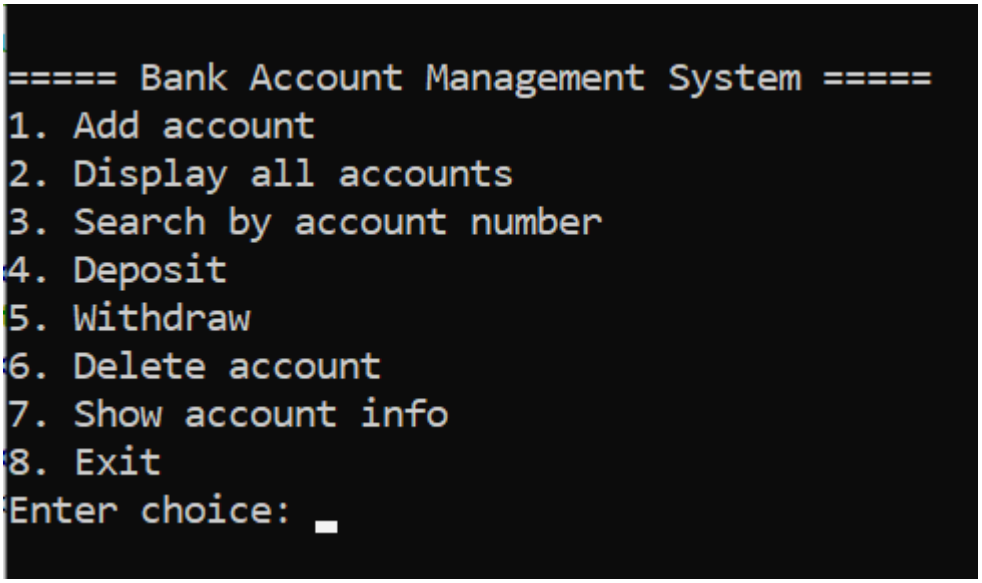
```
// withdrawal process with comprehensive error handling
bool performWithdrawal(BankSystem& bankSystem) {
    while (true) {
        string accNum = getAccountNumber(bankSystem, true);
        if (accNum.empty()) return false;

        double amount = getAmount();
        if (amount == -1) return false;

        if (bankSystem.withdraw(accNum, amount)) {
            return true;
        } else {
            if (!askForRetry("withdrawal process")) {
                return false;
            }
            // Continue the loop to retry
        }
    }
}
```

Design Benefit: Users can immediately correct withdrawal amounts without re-entering account numbers or navigating menus.

1.8 Program Screenshots and Functionality

Main Menu Interface

The system provides a user-friendly menu-driven interface.



1. Adding New Accounts

```
Enter choice: 1

1. Basic Account
2. Savings Account
3. Checking Account
Enter type: 2
Enter account number (numbers only, min 3 digits): 1001
Enter customer name: John Smith
Enter initial balance: $1000
Enter interest rate (default 2.5%): 3.5
Account added successfully!
```

Account type selection features:

- Three account types with specific attributes
- Input validation for account numbers and names
- Prevention of duplicate account creation
- Custom interest rates for savings accounts
- Custom overdraft limits for checking accounts

2. Display All Accounts

```
Enter choice: 2

--- Account 1 ---
Account Type: Savings Account
Account Number: 1001
Customer Name:  John Smith
Balance: $1000
Interest Rate: 3.5%

--- Account 2 ---
Account Type: Checking Account
Account Number: 1002
Customer Name: Jane Doe
Balance: $500
Overdraft Limit: $300
```

Account listing display functionality provides comprehensive account information.

3. Search Functionality

```
Enter choice: 3
Enter account number: 1001

Account found:
Account Type: Savings Account
Account Number: 1001
Customer Name:  John Smith
Balance: $1000
Interest Rate: 3.5%
```

Search results display detailed account information.

4. Deposit Operations

```
Enter choice: 4
Enter account number: 1001
Enter deposit amount: $250
Deposit successful! Previous: $1000, Deposited: $250, New: $1250
```

Successful deposit handling:

- Validates positive amounts
- Checks account existence
- Clear feedback on transaction success

5. Withdrawal Operations

Withdrawal with overdraft protection:

- Handles insufficient funds errors
- Provides clear error messaging

```
Enter choice: 5
Enter account number (numbers only, min 3 digits): 1001
Enter amount: $2000
Error: Insufficient funds! Available: $1254
Error in withdrawal process. Do you want to retry? (y/n):
```

```
Enter choice: 5
Enter account number (numbers only, min 3 digits): 1002
Enter amount: $700
Error: Insufficient funds! Available: $100
Error in withdrawal process. Do you want to retry? (y/n):
```

Error message for invalid numbers

```
Enter choice: 5
Enter account number (numbers only, min 3 digits): 1001
Enter amount: $abc
Error: Invalid input. Please enter a valid number.
Error in amount input. Do you want to retry? (y/n): y
Enter amount: $@
Error: Invalid input. Please enter a valid number.
Error in amount input. Do you want to retry? (y/n): _
```

6. Delete Account Display

Account deletion with confirmation (Y stands for yes, N stands for no).

7. Account Information Display

Detailed account info with special features display.

Advanced Features for Q1

1. Complete Inheritance Implementation

The system goes beyond basic requirements by implementing a complete inheritance hierarchy:

SavingsAccount with Interest Calculations:

```
class SavingsAccount : public Account {
private:
    double interestRate;
public:
    SavingsAccount(const string& accNum = "", const string& custName = "",
                   double initialBalance = 0.0, double rate = 2.5)
        : Account(accNum, custName, initialBalance), interestRate(rate) {}

    void showSpecialFeatures() const override {
```

```
        cout << "Annual interest on current balance: $" << balance * (interestRate / 100)
             << "\nInterest Rate: " << interestRate << "%" << endl;
    }
};
```

CheckingAccount with Overdraft Protection:

```
class CheckingAccount : public Account {
private:
    double overdraftLimit;
public:
    CheckingAccount(const string& accNum = "", const string& custName = "",
                    double initialBalance = 0.0, double overdraft = 500.0)
        : Account(accNum, custName, initialBalance), overdraftLimit(overdraft) {}

    bool canWithdraw(double amount) const override {
        return amount <= (balance + overdraftLimit);
    }
};
```

Design Benefits:

- Code reusability through common base functionality
- Polymorphic behavior allows uniform treatment of different account types
- Easy extension for new account types without modifying existing code
- Virtual functions enable runtime dispatch for account-specific operations

2. File Persistence System/Implementation

Advanced Data Management with Automatic Save/Load:

```
void loadAccountsFromFile() {
    ifstream inFile(DATA_FILE);
    if (!inFile) {
        cout << "No existing data found. Starting fresh." << endl;
        return;
    }

    string line, type, number, name;
    double balance, rate, limit;
    int count = 0;

    while (getline(inFile, line) && line.substr(0, 5) == "TYPE:") {
        type = line.substr(5);
```

```
        // Read account data
        if (getline(inFile, line) && line.substr(0, 7) == "NUMBER:")
            number = line.substr(7);
        if (getline(inFile, line) && line.substr(0, 5) == "NAME:")
            name = line.substr(5);
        if (getline(inFile, line) && line.substr(0, 8) == "BALANCE:")
            balance = stod(line.substr(8));

        // Create appropriate account type
        unique_ptr<Account> acc;
        if (type == "Basic Account") {
            acc = make_unique<Account>(number, name, balance);
        } else if (type == "Savings Account") {
            if (getline(inFile, line) && line.substr(0, 14) == "INTEREST_RATE:")
                rate = stod(line.substr(14));
            acc = make_unique<SavingsAccount>(number, name, balance, rate);
        } else if (type == "Checking Account") {
            if (getline(inFile, line) && line.substr(0, 16) == "OVERDRAFT_LIMIT:")
                limit = stod(line.substr(16));
            acc = make_unique<CheckingAccount>(number, name, balance, limit);
        }

        if (acc) {
            addAccount(std::move(acc));
            count++;
        }
        getline(inFile, line); // Skip separator
    }
    cout << count << " accounts loaded." << endl;
}
```

Features:

- **Automatic Data Recovery**: System loads previous session data on startup
- **Multiple Account Type Support**: Correctly handles different account types in files
- **Structured File Format**: Uses tagged format for reliable data parsing
- **Error Handling**: Graceful handling of corrupted or missing files
- **Session Management**: Automatically saves data on program exit

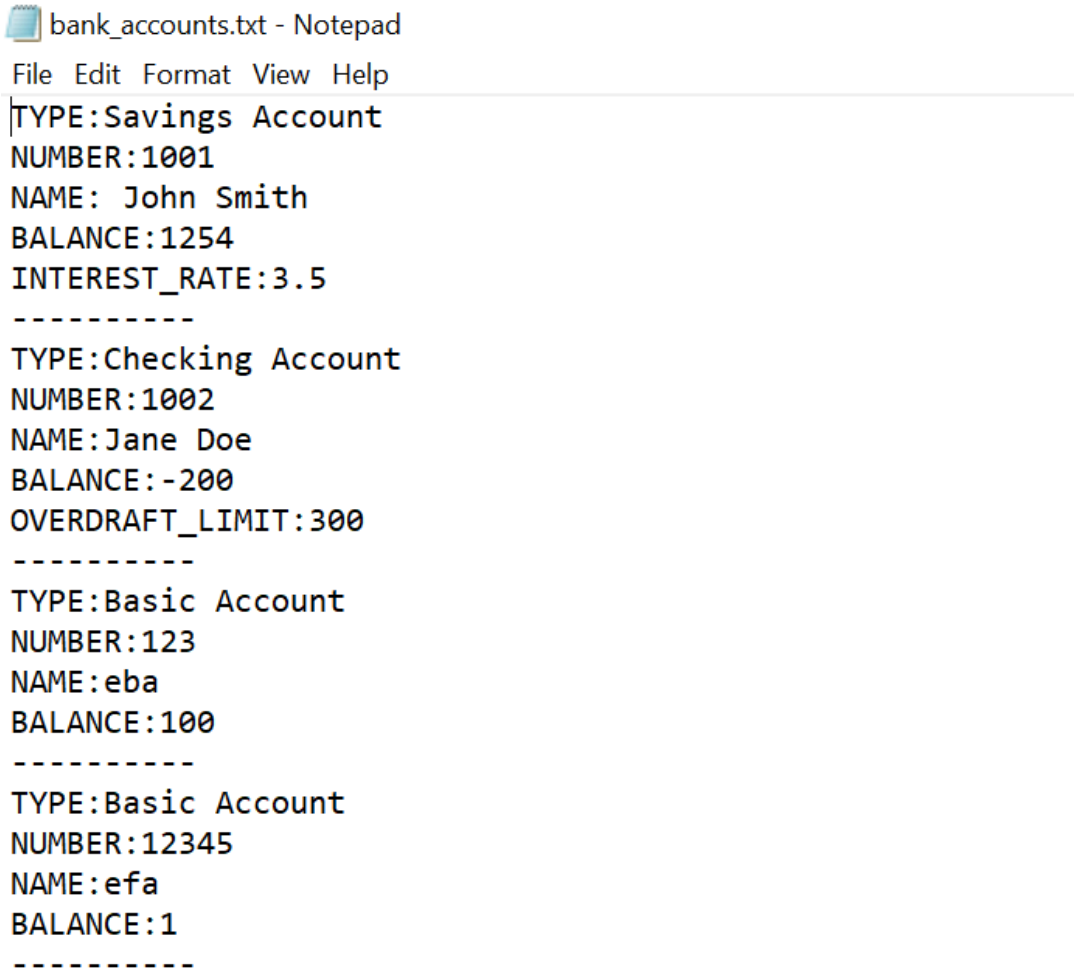
Important note: Do not forget to exit properly (option 8) before closing the program so all your accounts can be saved.

```
Enter choice: 8
Thank you for using Bank Account Management System!
4 accounts saved.
```

```
4 accounts loaded.

===== Bank Account Management System =====
1. Add account
2. Display all accounts
3. Search by account number
4. Deposit
5. Withdraw
6. Delete account
7. Show account info
8. Exit
Enter choice:
```

Text file to save all the previous accounts:

```
class BankSystem {
private:
    Node* head = nullptr;
    const string DATA_FILE = "./bank_accounts.txt";
```



3. Professional Features

- **Account type-specific behaviors** (interest rates, overdrafts)
- **Detailed account information displays** with specialized features
- **Confirmation prompts** for destructive operations (account deletion)
- **Professional error messaging** and user feedback
- **Comprehensive input validation** with immediate retry capabilities

Question 2: Warehouse Inventory and Shipping System Using Stack and Queue

2.1 System Overview

The warehouse system models real inventory management where:

- **Incoming items** are processed LIFO (most recent first)
- **Shipping queue** operates FIFO (first processed, first shipped)

This reflects real warehouse operations where recently received items are often processed first while maintaining shipping order fairness.

2.2 Custom Data Structure Implementation

Template-Based Design Strategy

Why Templates: Provides type safety and reusability while maintaining performance. The same Stack/Queue can handle any data type without code duplication.

Template Stack Implementation

```
template <typename T>
class Stack {
    struct Node {
        T data;
        Node* next;
        Node(const T& d, Node* n = nullptr) : data(d), next(n) {}
    };
    Node* topNode;
public:
    Stack() : topNode(nullptr) {}
    ~Stack() { while (!empty()) pop(); }

    void push(const T& val) { topNode = new Node(val, topNode); }
    void pop() {
        if (topNode) {
            Node* temp = topNode;
            topNode = topNode->next;
            delete temp;
        }
    }
    T& top() {
        if (!topNode) throw runtime_error("Stack is empty");
        return topNode->data;
    }
    bool empty() const { return topNode == nullptr; }

    // For listing all items (top to bottom, non-destructive)
```

```
    void toVector(vector<T>& out) const {
        Node* curr = topNode;
        while (curr) {
            out.push_back(curr->data);
            curr = curr->next;
        }
    }

    // For persistence: extract all items (top to bottom) into a vector
    void extractToVector(vector<T>& out) {
        vector<T> temp;
        while (!empty()) {
            temp.push_back(top());
            pop();
        }
        // Reverse to get bottom-to-top order
        for (auto it = temp.rbegin(); it != temp.rend(); ++it)
            out.push_back(*it);
    }
};
```

Single Pointer Design Benefits: Using only topNode maintains O(1) insertion and removal operations while keeping the implementation simple and efficient.

Queue Implementation with Dual Pointers

```
template <typename T>
class Queue {
    struct Node {
        T data;
        Node* next;
        Node(const T& d) : data(d), next(nullptr) {}
    };
    Node *frontNode, *backNode;
public:
    Queue() : frontNode(nullptr), backNode(nullptr) {}
    ~Queue() { while (!empty()) pop(); }

    void push(const T& val) {
        Node* n = new Node(val);
        if (backNode) backNode->next = n;
```

```
        backNode = n;
        if (!frontNode) frontNode = n;
    }

    void pop() {
        if (frontNode) {
            Node* temp = frontNode;
            frontNode = frontNode->next;
            if (!frontNode) backNode = nullptr;
            delete temp;
        }
    }

    T& front() {
        if (!frontNode) throw runtime_error("Queue is empty");
        return frontNode->data;
    }

    bool empty() const { return frontNode == nullptr; }

    // For listing all items (front to back, non-destructive)
    void toVector(vector<T>& out) const {
        Node* curr = frontNode;
        while (curr) {
            out.push_back(curr->data);
            curr = curr->next;
        }
    }

    // For persistence: extract all items into a vector
    void extractToVector(vector<T>& out) {
        while (!empty()) {
            out.push_back(front());
            pop();
        }
    }
};
```

Dual Pointer Benefits: Maintains O(1) insertion and removal operations by avoiding traversal to find the back of the queue.

Item Management Class

```
class Item {
   string name;
public:
   Item(const string& n) : name(n) {}
   Item() : name("") {}
   string getName() const { return name; }
};
```

2.3 Warehouse Workflow Implementation

Complete Process Flow:

1. **Add Incoming Item** → Stack (LIFO storage)
2. **Process Item** → Move from Stack top to Queue rear
3. **Ship Item** → Remove from Queue front (FIFO shipping)

Key Operation Functions

Adding Items:

```
void addIncomingItem(Stack<Item>& inventory) {
   string item;
   cout << "Enter item name: ";
   getline(cin, item);
   if (item.empty()) {
      cout << "Item name cannot be empty.\n";
      return;
   }
   inventory.push(Item(item));
   cout << "Item \"" << item << "\" added to inventory.\n";
}
```

Processing Items:

```
void processIncomingItem(Stack<Item>& inventory, Queue<Item>& shippingQueue) {
   if (inventory.empty()) {
      cout << "No items in inventory to process.\n";
      return;
```

```
    }
    Item item = inventory.top(); // Most recent item (LIFO)
    inventory.pop();
    shippingQueue.push(item); // Add to shipping queue
    cout << "Processed \"" << item.getName()
         << "\" and added to shipping queue.\n";
}
```

Process Logic: Recent arrivals are prioritized for processing, but shipping maintains fairness through FIFO ordering.

Viewing Functions:

```
// View all items in the inventory stack (from top to bottom)
void viewAllIncomingItems(const Stack<Item>& inventory) {
    vector<Item> items;
    inventory.toVector(items);
    if (items.empty()) {
        cout << "No items in inventory.\n";
        return;
    }
    cout << "All items in inventory (top to bottom):\n";
    for (const auto& item : items)
        cout << "- " << item.getName() << endl;
}

// View all items in the shipping queue (from front to back)
void viewAllShippingItems(const Queue<Item>& shippingQueue) {
    vector<Item> items;
    shippingQueue.toVector(items);
    if (items.empty()) {
        cout << "No items in shipping queue.\n";
        return;
    }
    cout << "All items in shipping queue (front to back):\n";
    for (const auto& item : items)
        cout << "- " << item.getName() << endl;
}
```

Shipping Items:

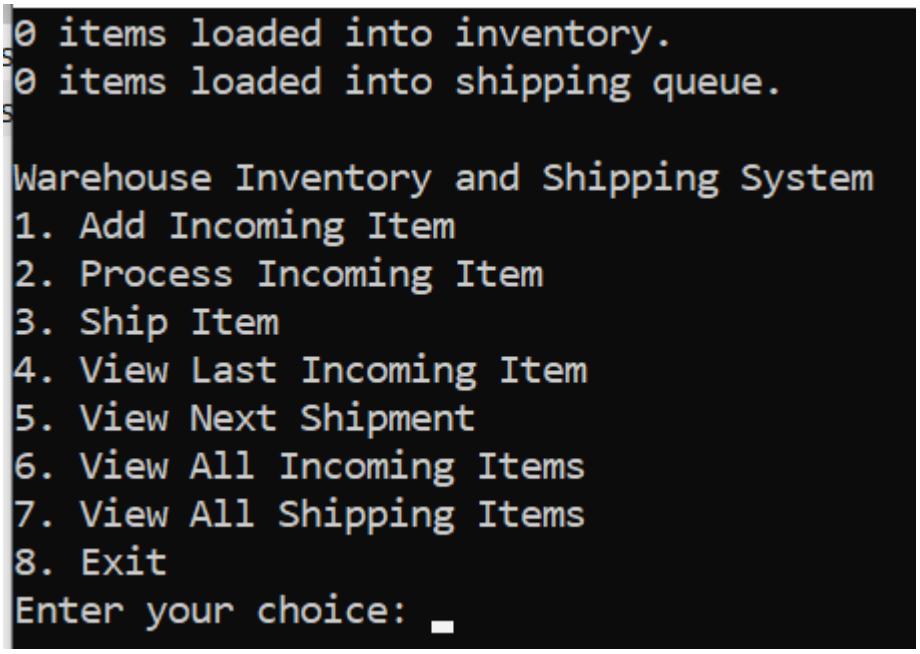
```
void shipItem(Queue<Item>& shippingQueue) {
    if (shippingQueue.empty()) {
        cout << "No items to ship.\n";
```

```
        return;
    }
    Item item = shippingQueue.front(); // Get first in line (FIFO)
    shippingQueue.pop();
    cout << "Shipping item: " << item.getName() << "\n";
}
```

2.4 Program Screenshots and Functionality

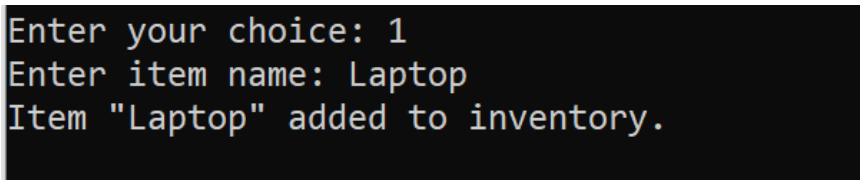
Main Menu Interface

User-friendly menu system for warehouse operations.



1. Adding Items to Inventory (Stack)

Stack Behavior: Items are added to the top (LIFO - Last In, First Out)



```
Enter your choice: 1
Enter item name: Monitor
Item "Monitor" added to inventory.
```

2. Viewing All Incoming Item

```
Enter your choice: 6
All items in inventory (top to bottom):
- Monitor
- Laptop
```

3. Viewing Last Incoming Item

View stack top functionality.

```
Enter your choice: 4
Last incoming item: Monitor
```

4. Processing Items (Stack to Queue)

Operation Flow: Items are removed from stack top and added to queue rear

```
Enter your choice: 2
Processed "Monitor" and added to shipping queue.
```

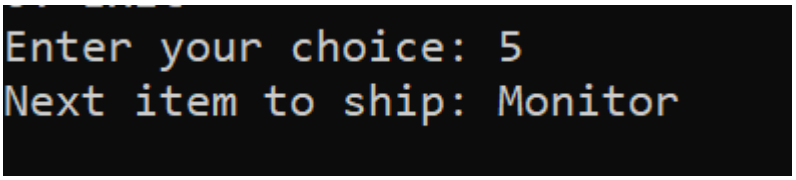
```
Enter your choice: 2
Processed "Laptop" and added to shipping queue.
```

5. Viewing All Shiping Item

```
Enter your choice: 7
All items in shipping queue (front to back):
- Monitor
- Laptop
```

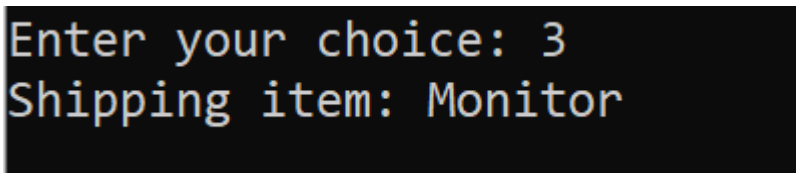
5. Viewing Next Shipment

View queue front functionality.



6. Shipping Items (Queue)

Queue Behavior: Items are removed from the front (FIFO - First In, First Out)



Complete Workflow Example

Step-by-step process:

1. Add Incoming Item → Laptop
2. Add Incoming Item → Monitor
    - Stack: [Monitor, Laptop] (top to bottom)
3. Process Incoming Item → Monitor moved to queue
    - Stack: [Laptop]
    - Queue: [Monitor]
4. Process Incoming Item → Laptop moved to queue
    - Stack: []
    - Queue: [Monitor, Laptop] (front to rear)
5. Ship Item → Monitor shipped
    - Queue: [Laptop]
6. Ship Item → Laptop shipped
    - Queue: []

2.5 Error Recovery Mechanisms

```
int choice;
do {
   // Menu display
   cout << "Enter your choice: ";
   if (!(cin >> choice)) {
```

```
        cin.clear();
        cin.ignore(10000, '\n');
        cout << "Invalid choice. Please enter a number between 1 and 6.\n";
        continue;
    }
    cin.ignore();

    switch (choice) {
        case 1:
            addIncomingItem(inventory);
            break;
        // ... other cases
        default:
            cout << "Invalid choice. Please try again.\n";
    }
} while (choice != 6);
```

Error Handling Features:

- **Empty Structure Checking:** Validates operations on empty stacks/queues
- **Input Validation:** Prevents empty item names
- **Exception Handling:** Graceful error recovery
- **User Feedback:** Clear status messages for all operations

Error Messages:

- Non-Numeric Input Error:

```
Warehouse Inventory and Shipping System
1. Add Incoming Item
2. Process Incoming Item
3. Ship Item
4. View Last Incoming Item
5. View Next Shipment
6. Exit
Enter your choice: hello
Invalid choice. Please enter a number between 1 and 6.

Warehouse Inventory and Shipping System
1. Add Incoming Item
2. Process Incoming Item
3. Ship Item
4. View Last Incoming Item
5. View Next Shipment
6. Exit
Enter your choice: @14
Invalid choice. Please enter a number between 1 and 6.
```

- Error for entering a valid number outside the range 1-6 (like 0, 7, 8, etc.)

```
Warehouse Inventory and Shipping System
1. Add Incoming Item
2. Process Incoming Item
3. Ship Item
4. View Last Incoming Item
5. View Next Shipment
6. Exit
Enter your choice: -1
Invalid choice. Please try again.

Warehouse Inventory and Shipping System
1. Add Incoming Item
2. Process Incoming Item
3. Ship Item
4. View Last Incoming Item
5. View Next Shipment
6. Exit
Enter your choice: 0
Invalid choice. Please try again.
```

- Error handling examples for various operations

```
Enter your choice: 2
No items in inventory to process.
```

```
Enter your choice: 4
No items in inventory.
```

```
Enter your choice: 5
No items in shipping queue.
```

2.6 Session Management

- **Automatic Save:** Data saved on program exit
- **Automatic Load:** Data restored on program startup
- **Graceful Handling:** Manages missing or corrupted files

2.7 Technical Implementation Highlights

Template Usage

- Generic classes work with any data type
- Type safety and code reusability
- Clean separation of data structure logic

Memory Management

- Proper cleanup in destructors
- Exception safety in operations
- Efficient memory usage

LIFO vs FIFO Demonstration

- **Stack:** Last item added is first to be processed
- **Queue:** First item processed is first to be shipped
- Clear separation of concerns between inventory and shipping

Advanced Features for q2

Question 2: Warehouse Inventory System - Advanced Features

1.File Persistence System

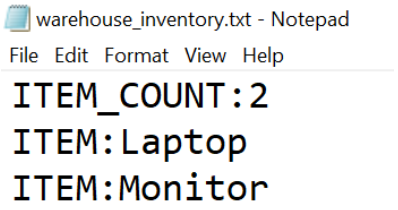
Advanced Data Persistence: The system implements sophisticated file persistence that maintains the exact order of items in both Stack and Queue structures across program sessions.

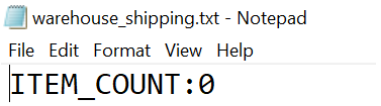
Persistence Architecture:

```
const string INVENTORY_FILE = "./warehouse_inventory.txt";
const string SHIPPING_QUEUE_FILE = "./warehouse_shipping.txt";
```

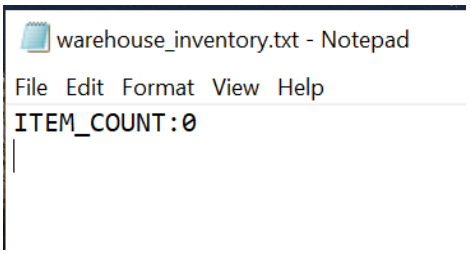
It gonna show you step by step of the process(but you need to save the info by exiting):

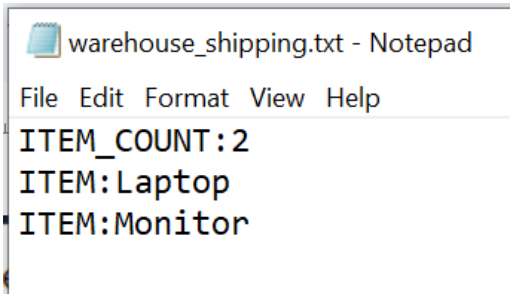
step1:



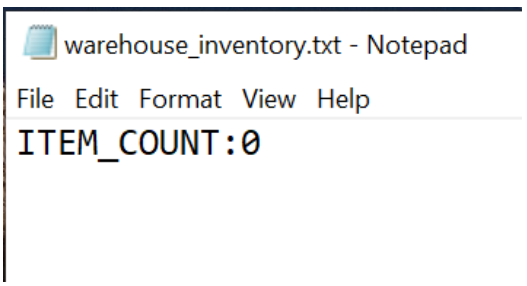


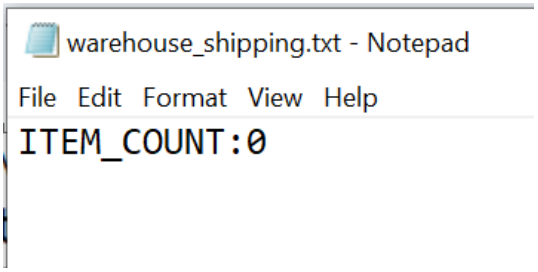
Step2:





Step3:





Stack Persistence Implementation

```
void saveInventory(Stack<Item> inventory) {
    ofstream outFile(INVENTORY_FILE);
    if (!outFile) {
        cerr << "Error: Could not open file for writing: " << INVENTORY_FILE << endl;
        return;
    }

    vector<Item> items;
    inventory.extractToVector(items); // Preserves LIFO order

    outFile << "ITEM_COUNT:" << items.size() << endl;
    for (const auto& item : items) {
        outFile << "ITEM:" << item.getName() << endl;
    }
    outFile.close();
    cout << items.size() << " items saved to inventory file." << endl;
}
```

Stack Loading with Order Preservation

```
void loadInventory(Stack<Item>& inventory) {
    ifstream inFile(INVENTORY_FILE);
    if (!inFile) {
        cout << "No existing inventory data found." << endl;
        return;
    }
```

```
    // Clear existing inventory
    while (!inventory.empty()) {
        inventory.pop();
    }

    try {
        string line;
        int count = 0;
        getline(inFile, line);
        if (line.substr(0, 11) == "ITEM_COUNT:") {
            count = stoi(line.substr(11));
        }

        vector<Item> items;
        for (int i = 0; i < count; i++) {
            getline(inFile, line);
            if (line.substr(0, 5) == "ITEM:") {
                string itemName = line.substr(5);
                items.push_back(Item(itemName));
            }
        }

        // CRITICAL: Load items in reverse order to maintain LIFO structure
        for (auto it = items.rbegin(); it != items.rend(); ++it) {
            inventory.push(*it);
        }

        cout << count << " items loaded into inventory." << endl;
    } catch (const exception& e) {
        cerr << "Error loading inventory data: " << e.what() << endl;
    }
}
```

Why Reverse Order Loading:

- **LIFO Preservation**: Items saved bottom-to-top must be loaded top-to-bottom
- **Stack Semantics**: Maintains the same stack order across program sessions
- **Correct Behavior**: Last item added before save becomes top item after load

Queue Persistence Implementation

```
void saveShippingQueue(Queue<Item> shippingQueue) {
    ofstream outFile(SHIPPING_QUEUE_FILE);
    if (!outFile) {
        cerr << "Error: Could not open file for writing: " << SHIPPING_QUEUE_FILE <<
endl;
        return;
    }

    vector<Item> items;
    shippingQueue.extractToVector(items); // Preserves FIFO order

    outFile << "ITEM_COUNT:" << items.size() << endl;
    for (const auto& item : items) {
        outFile << "ITEM:" << item.getName() << endl;
    }
    outFile.close();
    cout << items.size() << " items saved to shipping queue file." << endl;
}
```

Queue Loading (Maintains FIFO Order)

```
void loadShippingQueue(Queue<Item>& shippingQueue) {
    // ... file reading code ...

    // Load items into queue in order (front to back)
    for (const auto& item : items) {
        shippingQueue.push(item);
    }
}
```

File Persistence for Data Structures

```
0 items loaded into inventory.
0 items loaded into shipping queue.
```

```
Enter your choice: 6
Exiting...
Saving data before exit...
2 items saved to inventory file.
0 items saved to shipping queue file.
Exiting...
```

```
2 items loaded into inventory.
0 items loaded into shipping queue.
```

```
Enter your choice: 6
Exiting...
Saving data before exit...
1 items saved to inventory file.
1 items saved to shipping queue file.
Exiting...
```

```
1 items loaded into inventory.
1 items loaded into shipping queue.
```

```
Exiting...
Saving data before exit...
0 items saved to inventory file.
1 items saved to shipping queue file.
Exiting...
```

```
0 items loaded into inventory.
1 items loaded into shipping queue.
```

```
Enter your choice: 6
Exiting...
Saving data before exit...
0 items saved to inventory file.
0 items saved to shipping queue file.
Exiting...
```

```
0 items loaded into inventory.
0 items loaded into shipping queue.
```

2. Template Benefits for This Use Case

Type Safety Benefits:

- **Compile-Time Checking:** Prevents accidentally mixing different item types
- **No Runtime Overhead:** Template instantiation happens at compile time
- **Code Reuse:** Same data structure code works for any type

Extensibility Benefits:

- **Easy Type Changes:** Can switch from Item to Product or any other type
- **Generic Operations:** Stack and Queue operations work regardless of data type
- **Future-Proof Design:** Adding new item properties doesn't require data structure changes

3. LIFO vs FIFO Demonstration

Stack Behavior (LIFO):

Add Items: A → B → C

Stack State: [C, B, A] (C on top)

Process Order: C → B → A (most recent first)

Queue Behavior (FIFO):

Add Items: A → B → C

Queue State: [A, B, C] (A at front)

Ship Order: A → B → C (first in, first out)

Real-World Application:

- **Inventory Processing:** Recent arrivals often need priority processing
- **Shipping Fairness:** Orders should be fulfilled in the order they were processed
- **Load Balancing:** Stack for quick access, Queue for fair distribution

Important note: Do not forget to exit properly (option 6) before closing the program so all your items can be saved.

Conclusion

Achievement Summary

Question 1: Bank Account Management System

Technical Accomplishments:

- Complete OOP implementation with inheritance, encapsulation, and polymorphism

- Hybrid memory management combining smart pointers for objects and raw pointers for structure
- Advanced error recovery system with immediate retry capabilities improving user experience
- Professional-grade input validation with context-aware error messages
- Robust file persistence supporting multiple account types with polymorphic serialization

Industry-Standard Features:

- Account type specialization (interest rates, overdraft protection)
- Transaction validation and security measures
- Data persistence across sessions with automatic save/load
- User-friendly error handling and recovery without menu disruption

Question 2: Warehouse Inventory System

Technical Accomplishments:

- Custom template-based Stack and Queue implementations with type safety
- Perfect LIFO/FIFO behavior demonstration with real-world applicability
- Advanced file persistence maintaining structure ordering across sessions
- Real-world workflow modeling (inventory → processing → shipping)
- Comprehensive error handling for edge cases and invalid operations

Professional Features:

- Generic data structures supporting any item type
- Session management with automatic save/load
- Clear separation between inventory and shipping concerns
- Intuitive menu interface with status feedback

Real-World Application Value

The implemented systems demonstrate understanding of:

- **Software Engineering Principles:** Clean, maintainable, extensible code
- **Industry Standards:** Professional error handling and user experience
- **Performance Considerations:** Appropriate algorithm and data structure choices
- **System Design:** Modular architecture supporting future enhancements

This comprehensive implementation showcases readiness for professional software development environments and provides a solid foundation for advanced system development.