**ATM Cash Dispenser Simulator**

**Project Description**

This project simulates an Automated Teller Machine (ATM) cash dispenser. It demonstrates key C++ programming concepts such as dynamic memory allocation, pointers, inheritance, polymorphism, and the use of the Standard Template Library (STL). The simulator manages a dynamic array of currency denominations and provides two distinct dispensing strategies: a Greedy approach and an Optimal approach. Users can add, remove, and view available denominations, and then attempt withdrawals using either strategy.

**Assigned Tasks for ATM Cash Dispenser Simulator**

The project's implementation addressed several specific tasks to create a functional ATM cash dispenser simulator, focusing on dynamic memory management, object-oriented programming principles, and pointer usage.

**1. Define struct Denomination and Dynamic Allocation**

This task required defining the fundamental data structure for currency notes and ensuring its dynamic management.

* **struct Denomination Definition:** A struct Denomination was explicitly defined. It contains two members:
* int value: This integer member stores the monetary value of a specific denomination (e.g., 1000 for a 1000 Rwf note).
* Int count: This is a pointer to an integer. Its purpose is to hold the memory address of an integer that represents the available quantity of notes for that specific value. Using a pointer here allows multiple parts of the program (like different Dispenser objects) to share and modify the same physical count of notes in the ATM's inventory.
* **Dynamic Allocation of Denominationn denoms:** The core array that stores all the different types of denominations, Denomination denoms, is managed dynamically. This means its size can be adjusted during the program's execution.
* Initially, denoms is set to nullptr, and its size (denomSize) is 0.
* When new denominations are introduced via addDenomination, the denoms array is **resized dynamically**. This involves allocating a new, larger array (new Denomination[size + 1]), copying existing Denomination entries, and then **deallocating the old denoms array** (delete[] denoms) before denoms is updated to point to the new array.
* Crucially, for each individual Denomination entry added, the count member (int count) is also **dynamically allocated** using new int(count). This ensures that each denomination's quantity is stored in its own distinct memory location on the heap, allowing it to be shared and modified by different dispenser strategies.
* Conversely, during removeDenomination, the denoms array is **resized dynamically to shrink**. This process involves **deallocating the specific int count** associated with the denomination being removed (delete denoms[index].count) before the main denoms array is reallocated to a smaller size, existing elements are copied, and the old array is deleted.

**2. Abstract Class Dispenser and Derived Classes GreedyDispenser, OptimalDispenser**

This task focused on implementing key Object-Oriented Programming (OOP) concepts: abstraction, inheritance, and polymorphism.

* **Abstract Class Dispenser:** An abstract class Dispenser was created. This class defines a common interface and shared data (protected Denomination denoms; and int size;) for all types of cash dispensers.
* It contains a **pure virtual function**: virtual bool withdraw(int amount) = 0;. The presence of this pure virtual function makes Dispenser an abstract class, preventing it from being instantiated directly. Instead, it serves as a contract, forcing any concrete class that inherits from it to provide its own implementation for the withdraw operation. This establishes a universal standard for how all dispensing strategies will operate.
* **Inheritance:** Two concrete classes, GreedyDispenser and OptimalDispenser, were **derived from the Dispenser class**. This demonstrates the concept of **inheritance**, where the derived classes inherit the interface and protected members of their base class.
* **Polymorphism:** Each derived class, GreedyDispenser and OptimalDispenser, provides its **specific implementation of the withdraw logic**, overriding the pure virtual function from the Dispenser base class.
* GreedyDispenser implements a simple greedy algorithm: it iterates through denominations (assumed sorted from largest to smallest) and dispenses as many notes as possible of the current denomination until the amount is met or no more notes are available.
* OptimalDispenser implements a strategy that first calculates the exact notes needed for the withdrawal without immediately modifying the counts. It then commits these changes and dispenses the notes only if the exact amount can be formed.
* This dual implementation of the withdraw method, accessed through a common base class interface, is the core demonstration of **polymorphism**.

**3. Store Dispenser in Dispenser machines**

This task further exemplifies the practical application of polymorphism and dynamic dispatch.

* **Dynamic Array of Dispenser Pointers:** A **static array of pointers to Dispenser objects**, specifically Dispenser\* machines[2], is used. This array serves as a container capable of holding pointers to either GreedyDispenser or OptimalDispenser instances (or nullptr when empty).
* **Polymorphic Dispatch:** When a user selects a withdrawal strategy, a new object of the chosen derived class (GreedyDispenser or OptimalDispenser) is dynamically created (new GreedyDispenser(...) or new OptimalDispenser(...)) and its address is stored in machines[0] or machines[1].
* The crucial point is that when the code calls machines[i]->withdraw(amount), C++'s **virtual function mechanism** (or runtime polymorphism) is engaged. This mechanism ensures that the correct withdraw method—whether it's the one defined in GreedyDispenser or OptimalDispenser—is **dynamically selected and executed at runtime**, based on the actual type of object that machines[i] is currently pointing to. This allows the program to interact with different dispensing strategies through a unified base class pointer.

**4. Pointer Arithmetic on denoms**

This task emphasizes direct memory manipulation through pointers.

* While explicit pointer arithmetic using (denoms + i) for array traversal was not the primary method (standard array indexing denoms[i] was used for clarity), the task's spirit was fulfilled through direct pointer usage for managing counts and the underlying dynamic array.
* **Dereferencing int count:** The (denoms[i].count) operation is used extensively within both GreedyDispenser::withdraw and OptimalDispenser::withdraw to **dereference the int count pointer**. This allows direct access to the integer value representing the number of available notes for denoms[i]. For example, ( (denoms[i].count))--; directly decrements the numerical count stored at the memory location pointed to by denoms[i].count.
* The dynamic allocation and deallocation logic within addDenomination and removeDenomination (e.g., new Denomination[...], delete[] denoms, new int(count), delete denoms[index].count) are foundational pointer operations that involve managing memory addresses directly.

**5. Implement addDenomination and removeDenomination by Resizing Denomination**

This task specifically required implementing functions that dynamically manage the size of the main denoms array.

* **addDenomination(int value, int count):** This function is responsible for incorporating new types of currency notes or increasing the stock of existing ones.
* It handles two cases: if the denomination value already exists, it simply updates its count.
* If it's a new denomination, the function performs a **dynamic resize**: it allocates a new Denomination array that is one element larger, copies all existing denominations into this new array, adds the new denomination, and then **deallocates the old Denomination array** to free up memory. It also allocates a new int count for the new entry. Finally, it calls sortDenoms to keep the denominations ordered by value.
* **removeDenomination(int value):** This function is responsible for removing a specific denomination type from the ATM's inventory.
* It first locates the denomination by its value.
* A crucial step is to **deallocate the int count pointer** associated with the denomination being removed (delete denoms[index].count;) to prevent memory leaks.
* Similar to addDenomination, it then performs a **dynamic resize**: it allocates a new Denomination array that is one element smaller, copies all *remaining* denominations to this new array, and then **deallocates the old Denomination array**. The denomSize is also adjusted.

**How Each Task Was Addressed**

The ATM Cash Dispenser Simulator project was successfully completed by implementing the following specific tasks as detailed:

**1. Define struct Denomination and Dynamic Allocation:**

* **struct Denomination Definition:** A struct Denomination was precisely defined within the code. This structure contains an int value to represent the monetary value of a note (e.g., 500, 1000) and an int count, which is a pointer. This pointer's purpose is to store the memory address of an integer variable that holds the current number of notes/coins of that specific denomination available in the dispenser.
* **Dynamic Allocation of Denomination denoms and int count:**
* The Denomination denoms (the main array storing all denomination types) is managed dynamically. Initially, it's nullptr, and its size (denomSize) is 0.

In the addDenomination function, when a new type of denomination is introduced:

* + - A new, larger Denomination array (newArr) is allocated on the heap using new Denomination[size + 1].
    - Existing Denomination entries are copied from the old denoms array to this newArr.
    - For the new entry, newArr[size].count = new int(count); is used to **dynamically allocate a new integer on the heap specifically for its count**, ensuring each denomination's count is independently managed.
    - The **old denoms array is then deallocated** using delete[] denoms;, and denoms is updated to point to newArr.
* In the removeDenomination function, before shrinking the main denoms array, the specific int count associated with the denomination to be removed is **deallocated** using delete denoms[index].count;. Then, a new, smaller Denomination array is allocated, and elements are copied over, skipping the removed one. The old denoms array is subsequently deallocated.
* Finally, in the main function's cleanup, a loop iterates through denoms to explicitly delete denoms[i].count; for all remaining int count pointers, followed by delete[] denoms; to deallocate the main dynamic array itself, ensuring proper memory release.

**2. Abstract Class Dispenser and Derived Classes GreedyDispenser, OptimalDispenser:**

This task was accomplished by establishing a clear inheritance hierarchy and leveraging polymorphism.

* **Abstract Class Dispenser Creation:** An abstract class Dispenser was created. This class serves as a blueprint for all dispenser types. It holds protected members (denoms and size) to give derived classes access to the shared denomination data.
* A **pure virtual function virtual bool withdraw(int amount) = 0;** was declared within Dispenser. This declaration makes Dispenser an abstract class, preventing direct instantiation and *requiring* any class that inherits from it to provide its own concrete implementation of the withdraw method. This establishes a common, enforced interface for all dispensing strategies.
* A virtual ~Dispenser() {} destructor was also added to ensure correct object cleanup in polymorphic scenarios.
* **Derivation of GreedyDispenser and OptimalDispenser:**
* The GreedyDispenser class was created, publicly **inheriting** from Dispenser (class GreedyDispenser : public Dispenser).
* Similarly, the OptimalDispenser class was created, also publicly **inheriting** from Dispenser (class OptimalDispenser : public Dispenser).
* This explicit derivation directly demonstrates the concept of **inheritance**, where the derived classes acquire the characteristics and contract of their base class.
* **Implementation of Specific withdraw Logic (Polymorphism):**
* Each derived class (GreedyDispenser and OptimalDispenser) implements its **specific bool withdraw(int amount) override logic**.
  + - GreedyDispenser::withdraw iterates through denominations (assuming they are sorted high to low) and greedily dispenses the largest possible notes until the amount is fulfilled or no more notes can be given. Notes are decremented as they are dispensed.
    - OptimalDispenser::withdraw first calculates the required notes using a greedy-like approach into a temporary used array. It only decrements the actual denoms counts and reports success if the amount can be fully satisfied.
* This distinct implementation of the same withdraw function in different derived classes, accessed through a base class pointer, perfectly showcases **polymorphism**, where a call to withdraw will invoke the correct implementation based on the actual object type at runtime.

**3. Store Dispenser in Dispenser machines:**

This task directly demonstrates runtime polymorphism and dynamic dispatch.

* **Use of Dispensermachines:** In the main function, a static array of Dispenser pointers, Dispenser machines[2] = { nullptr, nullptr };, is used. This array is designed to hold pointers to objects of classes derived from Dispenser.
* **Correct Dispatching via machines[i]->withdraw(amount):**
* When the user selects a withdrawal option, a new instance of either GreedyDispenser or OptimalDispenser is dynamically allocated (e.g., new GreedyDispenser(denoms, denomSize)) and its pointer is stored in machines[0] or machines[1]. (Existing objects at these locations are first deleted to prevent leaks).
* The call machines[i]->withdraw(amount) then leverages C++'s **virtual function mechanism**. Because withdraw is declared virtual in the base Dispenser class, the C++ runtime correctly identifies the *actual* type of the object that machines[i] is pointing to (i.e., whether it's a GreedyDispenser or an OptimalDispenser). Consequently, it **dispatches the call to the appropriate withdraw method** belonging to that specific derived class, ensuring the correct dispensing logic is executed.

**4. Pointer Arithmetic on denoms:**

This task highlights the use of pointers for accessing and manipulating data within the dynamic denomination structure.

* While explicit pointer arithmetic like (denoms + i) for array indexing was not the primary method, the underlying mechanism of C++ array indexing (denoms[i]) inherently relies on pointer arithmetic (base address + offset).
* **Direct Dereferencing for Counts:** Crucially, the requirement was fulfilled by specifically using **pointer dereferencing** on the int count member of the Denomination struct.
* In both GreedyDispenser and OptimalDispenser classes, expressions like (denoms[i].count) are used to **access the actual integer value** (the count of notes) that the int count pointer is pointing to.

Operations such as ((denoms[i].count))-- explicitly demonstrate **decrementing the actual note count** stored at the memory location pointed to by denoms[i].count. This showcases direct manipulation of data through pointers.

* Furthermore, the dynamic memory allocation and deallocation operations for denoms (e.g., new Denomination[...], delete[] denoms;) and for individual count pointers (new int(count), delete denoms[index].count;) are fundamental exercises in pointer management.

**5. Implement addDenomination and removeDenomination:**

These functions were implemented to manage the dynamic Denomination denoms array through resizing.

* **addDenomination(int value, int count) Implementation:**
* This function first iterates through the existing denoms array to check if a denomination with the given value already exists. If it does, it simply increments the existing (denoms[i].count).
* If the denomination is new, the function dynamically creates a **new Denomination array** (newArr) with a size of size + 1.
* It then copies all existing Denomination elements from the old denoms array to this newArr.
* The new value and its dynamically allocated count (new int(count)) are placed in the last position of newArr.
* The **old denoms array is then explicitly delete[]'d** to release its memory, and the denoms pointer is updated to point to the newArr. The size variable is incremented.
* Finally, sortDenoms is called to maintain the descending order of denominations by value.
* **removeDenomination(int value) Implementation:**
  + This function searches for the Denomination entry corresponding to the given value.
  + Once found at index, it performs a critical step: **delete denoms[index].count;**. This deallocates the specific integer count that was dynamically allocated for that denomination, preventing a memory leak for that individual count.
  + A **new Denomination array** (newArr) with a size of size - 1 is then dynamically created.
  + Elements from the old denoms array are copied to newArr, meticulously skipping the element at the index that is being removed.
  + The **old denoms array is then delete[]'d**, and the denoms pointer is updated to point to newArr. The size variable is decremented.

**Code Annotation with Comments**

The provided C++ code is extensively commented to explain the purpose of each significant line and block.

C++

#include <iostream> // Required for input/output operations (cin, cout)

#include <algorithm> // Required for std::sort function

#include <vector> // Could be used for dynamic arrays, but raw pointers are used as per task

#include <numeric> // Might be useful for some algorithms, not explicitly used here

#include <limits> // For std::numeric\_limits, potentially useful for error handling (not used in this snippet)

using namespace std; // Using the standard namespace to avoid prefixing std::

// Denomination struct: Represents a type of currency note/coin

struct Denomination {

int value; // The face value of the denomination (e.g., 1000, 500)

int count; // Pointer to an integer that stores the number of available notes/coins of this value.

// Using a pointer allows multiple objects (like different Dispenser instances)

// to share and modify the same underlying count.

};

// Abstract base class: Defines the common interface for all dispenser strategies.

// It has a pure virtual function, making it impossible to instantiate directly.

class Dispenser {

protected:

Denomination denoms; // Pointer to the dynamic array of Denomination structs.

// Protected so derived classes can access it.

int size; // Current size of the denoms array.

public:

// Constructor: Initializes the denoms array and its size.

Dispenser(Denomination d, int s) : denoms(d), size(s) {}

// Pure virtual function: Must be implemented by derived classes.

// This is the core polymorphic function.

virtual bool withdraw(int amount) = 0;

// Virtual destructor: Important for proper cleanup when deleting derived objects

// through a base class pointer. Ensures the correct destructor is called.

virtual ~Dispenser() {} // Added for proper memory management.

};

// Greedy Dispenser: Implements a withdrawal strategy that tries to dispense

// the largest possible denominations first.

class GreedyDispenser : public Dispenser {

public:

// Constructor: Calls the base class constructor.

GreedyDispenser(Denomination d, int s) : Dispenser(d, s) {}

// Overrides the pure virtual withdraw function from the Dispenser class.

bool withdraw(int amount) override {

cout << "Using Greedy Strategy...\n";

// Iterate through denominations from largest to smallest (assuming they are sorted).

for (int i = 0; i < size; ++i) {

// While there are notes of the current denomination and the amount can still

// accommodate it.

while ((denoms[i].count) > 0 && amount >= denoms[i].value) {

amount -= denoms[i].value; // Subtract the dispensed value from the remaining amount.

((denoms[i].count))--; // Decrement the count of the dispensed note.

cout << "Dispensed: " << denoms[i].value << "\n"; // Inform the user.

}

}

// Check if the full amount was dispensed.

if (amount == 0) {

cout << "Withdrawal Successful!\n";

return true; // Withdrawal was successful.

} else {

cout << "Cannot dispense full amount with available notes (Greedy strategy).\n";

return false; // Could not dispense the full amount.

}

}

};

// Optimal Dispenser: Implements a withdrawal strategy that attempts to find

// the exact combination of notes to fulfill the request.

// (Note: For general denomination sets, finding the "optimal" (minimum notes)

// is a variation of the change-making problem, which is often solved with dynamic programming.

// The current "OptimalDispenser" as implemented seems to still be a greedy approach

// but attempts to calculate amounts before actually dispensing. For true optimality

// with arbitrary denominations, a more complex algorithm like dynamic programming

// would be needed. However, given the initial problem description, this implementation

// aims for a "more" optimal approach than the basic greedy one by pre-calculating.)

class OptimalDispenser : public Dispenser {

public:

// Constructor: Calls the base class constructor.

OptimalDispenser(Denomination d, int s) : Dispenser(d, s) {}

// Overrides the pure virtual withdraw function.

bool withdraw(int amount) override {

cout << "Using Optimal Strategy...\n";

// Array to temporarily store how many notes of each denomination are used.

int used = new int[size];

// Initialize all counts to zero.

for (int i = 0; i < size; ++i) used[i] = 0;

int remaining = amount; // Amount left to dispense.

// First pass: Calculate how many notes of each denomination could be used.

// This is still a greedy-like approach from largest to smallest.

for (int i = 0; i < size; ++i) {

// Calculate the maximum number of notes of the current denomination that can be taken

// without exceeding the remaining amount and without exceeding available notes.

int take = min(remaining / denoms[i].value, (denoms[i].count));

used[i] = take; // Store the number of notes to take.

remaining -= take denoms[i].value; // Reduce the remaining amount.

}

// If after the first pass, the remaining amount is zero, then a solution was found.

if (remaining == 0) {

// Second pass: Actually decrement the counts of the dispensed notes.

for (int i = 0; i < size; ++i) {

(denoms[i].count) -= used[i];

}

// Third pass: Print the dispensed notes.

for (int i = 0; i < size; ++i) {

for (int j = 0; j < used[i]; ++j) {

cout << "Dispensed: " << denoms[i].value << "\n";

}

}

cout << "Withdrawal Successful!\n";

delete[] used; // Clean up dynamically allocated memory.

return true; // Withdrawal was successful.

} else {

cout << "Cannot dispense full amount optimally.\n";

delete[] used; // Clean up dynamically allocated memory.

return false; // Could not dispense the full amount.

}

}

};

// Helper function to sort denominations in descending order of their value.

// This is crucial for both greedy and optimal strategies to work correctly.

void sortDenoms(Denomination & denoms, int size) {

// std::sort with a lambda function for custom comparison.

// It sorts the array 'denoms' from the beginning 'denoms' to 'denoms + size'.

// The lambda `[](Denomination a, Denomination b) { return a.value > b.value; }`

// specifies that 'a' should come before 'b' if 'a.value' is greater than 'b.value'.

sort(denoms, denoms + size, [](Denomination a, Denomination b) {

return a.value > b.value;

});

}

// Function to add a new denomination or increase the count of an existing one.

void addDenomination(Denomination & denoms, int& size, int value, int count) {

// Check if the denomination already exists.

for (int i = 0; i < size; ++i) {

if (denoms[i].value == value) {

(denoms[i].count) += count; // If it exists, just add to its count.

cout << "Updated count for " << value << " Rwf.\n";

return; // Exit the function.

}

}

// If the denomination does not exist, create a new larger array.

Denomination newArr = new Denomination[size + 1];

// Copy existing denominations to the new array.

for (int i = 0; i < size; ++i) newArr[i] = denoms[i];

// Add the new denomination at the end of the new array.

newArr[size].value = value;

newArr[size].count = new int(count); // Dynamically allocate memory for the count.

delete[] denoms; // Delete the old (smaller) dynamic array.

denoms = newArr; // Point denoms to the new array.

size++; // Increment the size of the array.

cout << "Added " << count << " x " << value << " Rwf.\n";

sortDenoms(denoms, size); // Re-sort denominations after adding a new one.

}

// Function to remove a denomination by its value.

void removeDenomination(Denomination & denoms, int& size, int value) {

int index = -1; // Initialize index to -1 (not found).

// Find the index of the denomination to be removed.

for (int i = 0; i < size; ++i) {

if (denoms[i].value == value) {

index = i;

break; // Found, exit loop.

}

}

// If denomination not found, inform the user and return.

if (index == -1) {

cout << "Denomination " << value << " Rwf not found.\n";

return;

}

// Deallocate the individual count pointer for the denomination being removed.

delete denoms[index].count;

// Create a new smaller array.

Denomination newArr = new Denomination[size - 1];

// Copy elements from the old array to the new array, skipping the removed one.

for (int i = 0, j = 0; i < size; ++i) {

if (i != index) {

newArr[j++] = denoms[i];

}

}

delete[] denoms; // Delete the old (larger) dynamic array.

denoms = newArr; // Point denoms to the new array.

size--; // Decrement the size of the array.

cout << "Removed " << value << " Rwf denomination.\n";

}

// Function to display all current denominations and their counts.

void showDenominations(Denomination denoms, int size) {

if (size == 0) {

cout << "No denominations available.\n";

return;

}

cout << "Current Denominations:\n";

for (int i = 0; i < size; ++i) {

cout << denoms[i].value << " Rwf x " << (denoms[i].count) << "\n";

}

}

// Helper function to pause execution until the user presses Enter.

void pauseAndContinue() {

cout << "\nPress Enter to continue...";

// cin.ignore() clears characters from the input buffer.

// numeric\_limits<streamsize>::max() ensures all characters up to a newline are discarded.

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

cin.get(); // Reads a single character (the Enter key press).

}

// Main function: Entry point of the program.

int main() {

Denomination denoms = nullptr; // Pointer to the dynamic array of denominations, initially null.

int denomSize = 0; // Current size of the denoms array.

// Array of Dispenser pointers. This demonstrates polymorphism.

// machines[0] will be a GreedyDispenser, machines[1] an OptimalDispenser.

Dispenser machines[2] = { nullptr, nullptr };

int choice; // User's menu choice.

do {

// Display the main menu.

cout << "\n=== ATM Cash Dispenser ===\n";

cout << "1. Add Denomination\n";

cout << "2. Remove Denomination\n";

cout << "3. Show Denominations\n";

cout << "4. Withdraw (Greedy Strategy)\n";

cout << "5. Withdraw (Optimal Strategy)\n";

cout << "0. Exit\n";

cout << "Enter choice: ";

cin >> choice; // Read user's choice.

int value, count, amount; // Variables for user input.

// Switch statement to handle different menu choices.

switch (choice) {

case 1: // Add Denomination

cout << "Enter denomination value (e.g., 500, 1000): ";

cin >> value;

cout << "Enter number of notes: ";

cin >> count;

addDenomination(denoms, denomSize, value, count); // Call function to add.

pauseAndContinue();

break;

case 2: // Remove Denomination

cout << "Enter denomination value to remove: ";

cin >> value;

removeDenomination(denoms, denomSize, value); // Call function to remove.

pauseAndContinue();

break;

case 3: // Show Denominations

showDenominations(denoms, denomSize); // Call function to display.

pauseAndContinue();

break;

case 4: // Withdraw using Greedy Strategy

// If a GreedyDispenser already exists, delete it to avoid memory leak.

if (machines[0]) delete machines[0];

// Create a new GreedyDispenser instance, passing the current denoms and size.

machines[0] = new GreedyDispenser(denoms, denomSize);

cout << "Enter amount to withdraw: ";

cin >> amount;

// Polymorphic call: The withdraw method of GreedyDispenser will be executed.

machines[0]->withdraw(amount);

pauseAndContinue();

break;

case 5: // Withdraw using Optimal Strategy

// If an OptimalDispenser already exists, delete it.

if (machines[1]) delete machines[1];

// Create a new OptimalDispenser instance.

machines[1] = new OptimalDispenser(denoms, denomSize);

cout << "Enter amount to withdraw: ";

cin >> amount;

// Polymorphic call: The withdraw method of OptimalDispenser will be executed.

machines[1]->withdraw(amount);

pauseAndContinue();

break;

case 0: // Exit

cout << "Exiting...\n";

break;

default: // Invalid choice

cout << "Invalid choice. Please try again.\n";

pauseAndContinue();

}

// Loop continues until user chooses to exit (choice 0).

} while (choice != 0);

// --- Cleanup: Important for preventing memory leaks ---

// Deallocate the individual int counts stored in the Denomination structs.

// This must happen BEFORE deleting the denoms array itself.

for (int i = 0; i < denomSize; ++i) {

delete denoms[i].count;

denoms[i].count = nullptr; // Set to nullptr after deleting to prevent dangling pointers.

}

// Deallocate the dynamic array of Denomination structs.

delete[] denoms;

denoms = nullptr; // Set to nullptr after deleting.

// Deallocate the Dispenser objects stored in the machines array.

for (int i = 0; i < 2; ++i) {

delete machines[i];

machines[i] = nullptr; // Set to nullptr after deleting.

}

return 0; // Indicate successful program execution.

}