**Final Comprehensive Challenge: Inventory Management System**

**Your Task:**

Design and implement a simple Inventory Management System in C++ that meets the following requirements.

**1. Define a Structure for Inventory Items**

* Create a struct Item with the following members:
  + std::string name
  + int quantity
  + double price
* Provide a parameterized constructor for Item so that an object can be easily initialized (e.g., Item("Widget", 10, 1.99)).

**2. Create a Class to Manage the Inventory**

Define a class named Inventory that manages a dynamic array of Item objects. The class should include:

* **Data Members:**
  + A pointer to a dynamically allocated array of Item objects.
  + An integer that holds the current number of items.
  + An integer that holds the capacity of the array (to handle expansion if desired).
* **Member Functions:**
  + **Constructor:** A default constructor that initializes the dynamic array and sets initial values (e.g., a small starting capacity).
  + **Destructor:** A destructor that properly deallocates the dynamic array using delete[] to prevent memory leaks.
  + **Copy Constructor & Assignment Operator (Optional Advanced):** To ensure deep copies (if you choose to allow Inventory copying), overload these to avoid shallow copy issues.
  + **Add Item:** A function void addItem(const Item &newItem) that adds a new item to the inventory. (Hint: If your array is full, dynamically allocate a larger array and copy elements.)
  + **Remove Item:** A function bool removeItem(const std::string &itemName) that removes an item by its name. Return a boolean indicating success or failure.
  + **Update Quantity:** A function void updateQuantity(const std::string &itemName, int newQuantity) that updates the quantity of an item. *(Use pass-by-reference to locate and update the item within your array.)*
  + **Operator Overloading:**
    - Overload the operator[] so that the user can access an Item in the inventory by its index (e.g., inventory[0]).
    - Overload the insertion operator operator<< so that you can easily print details of an entire Inventory or even a single Item (if you prefer).

**3. Add Overloaded print Functions**

Implement at least two overloaded functions named print (outside or inside a class, as you see fit):

* One version should accept an integer (for example, to print the total number of items in the inventory).
* Another version should accept an Item (or even a list/array of items) and print its details in a neat, formatted style.

**4. Use References, Pointers, and Parameter Passing in main()**

In your main() function:

* **Create Instances:**
  + Create an instance of the Inventory class on the stack.
  + Dynamically allocate another Inventory object using a pointer.
* **Manipulate Data:**
  + Add several items to your inventory using your addItem function.
  + Update one of the item’s quantities by calling updateQuantity. Make sure to pass values (or items) by reference where applicable.
  + Demonstrate accessing an item with the overloaded operator[] and using a pointer or reference to modify one element directly.
  + Use your overloaded print functions to display:
    - The total number of items (as an integer).
    - The details of one or more items.
* **Clean Up:** Ensure that any dynamically allocated memory (for example, the second Inventory object) is properly deallocated using delete.

**5. Additional Requirements**

* **Memory Management:** Ensure that you use new/delete (or new[]/delete[]) correctly.
* **References vs. Pointers:** Use references for function parameters when you wish to modify objects directly, and pointers when you need to manage dynamic allocation.
* **Operator and Function Overloading:** Your overloaded operators (such as operator<< and operator[]) and overloaded functions (the print overloads) should follow intuitive behavior similar to built-in types.
* **Code Readability:** Write clear, commented code and use proper formatting.

**Your Challenge Summary**

> **Implement a complete Inventory Management System in C++ that:** > > - Uses a structure (Item) with a parameterized constructor. > - Manages a dynamic array of items inside a class (Inventory) with a constructor, destructor, and (optionally) copy control methods. > - Provides functions to add, remove, and update items. > - Properly overloads operators ([] for array access, and << for output). > - Implements overloaded print functions for different data types. > - Demonstrates the use of references, pointers, and proper parameter passing in main(). > - Ensures no memory leaks via proper dynamic memory management.

#include <iostream>

#include <string>

using namespace std;

//---------------------------

// Part 1: Define the Structure for Inventory Items

//---------------------------

struct Item {

    string name;

    int quantity;

    double price;

    // Parameterized Constructor: Allows easy initialization like Item("Widget", 10, 1.99)

    Item(const string& itemName, int itemQuantity, double itemPrice)

        : name(itemName), quantity(itemQuantity), price(itemPrice) {}

    // Default Constructor: Required when allocating an array of Items.

    Item() : name(""), quantity(0), price(0.0) {}

};

//---------------------------

// Part 2: Inventory Class for Managing Items

//---------------------------

class Inventory {

private:

    Item\* items;       // Dynamic array of Item objects

    int itemCount;     // Current number of items stored

    int capacity;      // Maximum capacity of our dynamic array

    // Private helper function to double the capacity when the array is full.

    void resize() {

        int newCapacity = capacity \* 2;

        Item\* newItems = new Item[newCapacity];

        // Copy existing items into the new array.

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

            newItems[i] = items[i];

        }

        delete[] items;   // Free the old memory

        items = newItems; // Point to the newly allocated array

        capacity = newCapacity;

    }

public:

    // Default constructor: Sets up an initial small capacity.

    Inventory() : itemCount(0), capacity(2) {

        items = new Item[capacity];

    }

    // Destructor: Ensures we deallocate the dynamic memory to prevent memory leaks.

    ~Inventory() {

        delete[] items;

    }

    // Copy Constructor (Advanced): Performs a deep copy to keep separate copies of data.

    Inventory(const Inventory& other) : itemCount(other.itemCount), capacity(other.capacity) {

        items = new Item[capacity];

        for (int i = 0; i < itemCount; ++i)

            items[i] = other.items[i];

    }

    // Assignment Operator (Advanced): Ensures deep copy and proper memory management.

    Inventory& operator=(const Inventory& other) {

        if (this != &other) {

            delete[] items; // Free current memory

            itemCount = other.itemCount;

            capacity = other.capacity;

            items = new Item[capacity];

            for (int i = 0; i < itemCount; ++i)

                items[i] = other.items[i];

        }

        return \*this;

    }

    // Add Item: Adds a new item to the inventory. Resizes the array if needed.

    void addItem(const Item& newItem) {

        if (itemCount == capacity) {

            resize();

        }

        items[itemCount++] = newItem;

    }

    // Remove Item: Removes an item by name and returns true if successful.

    bool removeItem(const string& itemName) {

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

            if (items[i].name == itemName) {

                // Shift the array to remove the gap.

                for (int j = i; j < itemCount - 1; ++j) {

                    items[j] = items[j + 1];

                }

                --itemCount;

                return true;

            }

        }

        return false; // Item not found.

    }

    // Update Quantity: Searches by item name and updates the quantity.

    void updateQuantity(const string& itemName, int newQuantity) {

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

            if (items[i].name == itemName) {

                items[i].quantity = newQuantity;

                return;

            }

        }

        cerr << "Item \"" << itemName << "\" not found in inventory.\n";

    }

    // Overloaded operator[]: Enables array-like access to inventory items.

    Item& operator[](int index) {

        if (index >= 0 && index < itemCount)

            return items[index];

        throw out\_of\_range("Index out of range");

    }

    // Overloaded operator<<: Allows easy printing of the entire inventory.

    friend ostream& operator<<(ostream& os, const Inventory& inventory) {

        os << "Inventory details:\n";

        for (int i = 0; i < inventory.itemCount; ++i) {

            os << "Item " << i + 1 << ":\n";

            os << "  Name: " << inventory.items[i].name << "\n";

            os << "  Quantity: " << inventory.items[i].quantity << "\n";

            os << "  Price: $" << inventory.items[i].price << "\n";

        }

        return os;

    }

    // Getter for total number of items, which will be used by one of our print functions.

    int getItemCount() const {

        return itemCount;

    }

};

//---------------------------

// Part 3: Overloaded Print Functions

//---------------------------

// Print function that accepts an integer (e.g., total number of items)

void print(int totalItems) {

    cout << "Total number of items: " << totalItems << "\n";

}

// Print function that accepts a single Item and prints its details in a formatted style.

void print(const Item& item) {

    cout << "Item Details:\n";

    cout << "  Name: " << item.name << "\n";

    cout << "  Quantity: " << item.quantity << "\n";

    cout << "  Price: $" << item.price << "\n";

}

//---------------------------

// Part 4: main() Using References, Pointers, and Parameter Passing

//---------------------------

int main() {

    // Create an Inventory instance on the stack.

    Inventory inventory;

    // Dynamically allocate an Inventory object using a pointer.

    Inventory\* inventoryPtr = new Inventory();

    // -----------------------------

    // Manipulate Data

    // -----------------------------

    // Adding several items using addItem function.

    inventory.addItem(Item("Widget", 10, 1.99));

    inventory.addItem(Item("Gadget", 5, 4.99));

    inventory.addItem(Item("Thingamajig", 20, 2.49));

    // Add items to the dynamically allocated Inventory.

    inventoryPtr->addItem(Item("Doohickey", 15, 3.99));

    inventoryPtr->addItem(Item("Contraption", 8, 9.99));

    // Update the quantity of a specific item.

    inventory.updateQuantity("Widget", 12);

    // -----------------------------

    // Demonstrate Access with Overloaded operator[]

    // -----------------------------

    try {

        // Use the overloaded operator[] to access the first item.

        Item& firstItem = inventory[0];

        // Modify the item's quantity directly using the reference.

        firstItem.quantity = 100;

        cout << "\nAfter modifying the first item using operator[]:\n";

        print(firstItem);  // Use the print overload for a single Item.

    } catch (const out\_of\_range& ex) {

        cerr << ex.what() << "\n";

    }

    // -----------------------------

    // Use Overloaded Print Functions

    // -----------------------------

    // Print the total number of items in the inventory.

    print(inventory.getItemCount());

    // Use the overloaded << operator to print the entire inventory.

    cout << "\nInventory (stack instance):\n" << inventory;

    // Print the dynamically allocated inventory.

    cout << "\nInventory (dynamically allocated instance):\n" << \*inventoryPtr;

    // -----------------------------

    // Clean Up

    // -----------------------------

    delete inventoryPtr;  // Free the dynamically allocated Inventory object.

    return 0;

}

**1. The Item Structure**

struct Item {

    string name;

    int quantity;

    double price;

    // Parameterized Constructor

    Item(const string& itemName, int itemQuantity, double itemPrice)

        : name(itemName), quantity(itemQuantity), price(itemPrice) {}

    // Default Constructor: This is needed for creating arrays of Items.

    Item() : name(""), quantity(0), price(0.0) {}

};

**Explanation:**

* **Data Members:**
  + name: A string that holds the name of the item.
  + quantity: An integer that stores how many units are available.
  + price: A double representing the cost per unit.
* **Constructors:**
  + **Parameterized Constructor:** Allows you to quickly create an item with specific values. For example, Item("Widget", 10, 1.99) will create an item named "Widget" with a quantity of 10 and a price of 1.99.
  + **Default Constructor:** This constructor is important when you create a dynamic array (e.g., new Item[capacity]). The array needs to initialize each element, and without a default constructor, the compiler wouldn’t know how to create an empty Item.

**2. The Inventory Class**

This class manages a collection of Item objects using a dynamic array. It handles adding, removing, updating items, and even expanding the array if needed.

class Inventory {

    private:

        Item\* items;       // Pointer to a dynamically allocated array of Item objects.

        int itemCount;     // Current number of items stored.

        int capacity;      // Maximum capacity of our dynamic array.

        // Private helper function to resize the array when it’s full.

        void resize() {

            int newCapacity = capacity \* 2;      // We double the capacity.

            Item\* newItems = new Item[newCapacity]; // Allocate a new array with larger capacity.

            // Copy each current item to the new array.

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

                newItems[i] = items[i];

            }

            delete[] items;  // Free up the old memory.

            items = newItems;  // Point to the new array.

            capacity = newCapacity;  // Update the capacity.

        }

    public:

        // Default constructor: Initializes with a small starting capacity.

        Inventory() : itemCount(0), capacity(2) {

            items = new Item[capacity];

        }

        // Destructor: Cleans up dynamic memory to prevent memory leaks.

        ~Inventory() {

            delete[] items;

        }

        // Copy Constructor (Advanced): Creates a new Inventory with the same items.

        Inventory(const Inventory& other) : itemCount(other.itemCount), capacity(other.capacity) {

            items = new Item[capacity];

            for (int i = 0; i < itemCount; ++i)

                items[i] = other.items[i];

        }

        // Assignment Operator (Advanced): Allows one inventory to be assigned to another.

        Inventory& operator=(const Inventory& other) {

            if (this != &other) {

                delete[] items;  // Free current memory.

                itemCount = other.itemCount;

                capacity = other.capacity;

                items = new Item[capacity];

                for (int i = 0; i < itemCount; ++i)

                    items[i] = other.items[i];

            }

            return \*this;

        }

        // Add Item: Adds a new item to our inventory.

        void addItem(const Item& newItem) {

            if (itemCount == capacity) {  // Check if the array is full.

                resize();  // Double the capacity if needed.

            }

            items[itemCount++] = newItem;  // Add the new item and increment itemCount.

        }

        // Remove Item: Searches for an item by name and removes it.

        bool removeItem(const string& itemName) {

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

                if (items[i].name == itemName) {

                    // If found, shift all following items one place to the left.

                    for (int j = i; j < itemCount - 1; ++j) {

                        items[j] = items[j + 1];

                    }

                    --itemCount;  // Reduce the count.

                    return true;  // Successfully removed.

                }

            }

            return false;  // Item not found.

        }

        // Update Quantity: Finds an item by name and updates its quantity.

        void updateQuantity(const string& itemName, int newQuantity) {

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

                if (items[i].name == itemName) {

                    items[i].quantity = newQuantity;  // Set the new quantity.

                    return;

                }

            }

            cerr << "Item \"" << itemName << "\" not found in inventory.\n";

        }

        // Overloaded operator[]: Provides array-like access to the items.

        Item& operator[](int index) {

            if (index >= 0 && index < itemCount)

                return items[index];

            throw out\_of\_range("Index out of range");

        }

        // Overloaded operator<<: Allows us to output the entire inventory easily.

        friend ostream& operator<<(ostream& os, const Inventory& inventory) {

            os << "Inventory details:\n";

            for (int i = 0; i < inventory.itemCount; ++i) {

                os << "Item " << i + 1 << ":\n";

                os << "  Name: " << inventory.items[i].name << "\n";

                os << "  Quantity: " << inventory.items[i].quantity << "\n";

                os << "  Price: $" << inventory.items[i].price << "\n";

            }

            return os;

        }

        // Getter function to access the current count of items.

        int getItemCount() const {

            return itemCount;

        }

    };

**Detailed Breakdown:**

* **Private Data Members:**
  + items: This pointer holds the address of a dynamic array of Item objects.
  + itemCount: Keeps track of how many items are currently stored.
  + capacity: Represents the total number of Item objects the array can hold before needing to expand.
* **The** resize() **Function:**
  + When we try to add an item and the array is full, we need more space.
  + We double the capacity, allocate a new array of Item objects with the new capacity, copy existing items, delete the old array, and update our pointer and capacity.
* **Constructors and Destructor:**
  + **Default Constructor:** Initializes itemCount to 0, sets an initial capacity (in this case, 2), and allocates an array of Items.
  + **Destructor:** Uses delete[] to deallocate the dynamic array, preventing memory leaks.
* **Copy Constructor & Assignment Operator (Advanced):** These ensure that if you copy an Inventory object, you get an independent copy instead of a shallow copy (which would share the same memory). They allocate new memory and copy all elements from the source object.
* **Member Functions Explained:**
  + addItem()**:** When you call this function with a new Item, it checks if the current array is full. If it is, the array is resized. Then, the new item is placed into the array, and the count increases.
  + removeItem()**:** This function loops over the items to find a match by name. Once found, it shifts all later elements one position to the left (thus "removing" the item) and then decrements itemCount.
  + updateQuantity()**:** Searches for the item by name and updates its quantity field. If the item is not found, it prints an error message.
  + operator[]**:** Lets you use the bracket syntax (like inventory[0]) to access an item. It returns a reference, so you can modify the item directly.
  + operator<<**:** A friend function that overloads the insertion operator. It makes printing the entire inventory easy by sending your inventory details directly into an output stream (like cout).
* **Getter Function:** getItemCount() returns the current number of items, which is useful for printing or other operations.

**3. Overloaded Print Functions**

Outside the class, we have two versions of the print function that let us output information in different formats:

// Print function that accepts an integer (e.g., the total number of items)

void print(int totalItems) {

    cout << "Total number of items: " << totalItems << "\n";

}

// Print function that accepts a single Item and prints its details.

void print(const Item& item) {

    cout << "Item Details:\n";

    cout << "  Name: " << item.name << "\n";

    cout << "  Quantity: " << item.quantity << "\n";

    cout << "  Price: $" << item.price << "\n";

}

**Explanation:**

* print(int totalItems)**:** Simply prints the total count of items.
* print(const Item& item)**:** Takes an Item (passed by reference to avoid unnecessary copies) and outputs its details in a formatted and readable style.

**4. The main() Function**

This function ties everything together—creating inventory objects, manipulating data, and printing the results.

int main() {

    // Create an Inventory instance on the stack.

    Inventory inventory;

    // Dynamically allocate an Inventory object using a pointer.

    Inventory\* inventoryPtr = new Inventory();

    // -----------------------------

    // Adding Items to Inventory

    // -----------------------------

    // Adding items to the inventory on the stack.

    inventory.addItem(Item("Widget", 10, 1.99));

    inventory.addItem(Item("Gadget", 5, 4.99));

    inventory.addItem(Item("Thingamajig", 20, 2.49));

    // Adding items to the dynamically allocated inventory.

    inventoryPtr->addItem(Item("Doohickey", 15, 3.99));

    inventoryPtr->addItem(Item("Contraption", 8, 9.99));

    // -----------------------------

    // Updating an Item's Quantity

    // -----------------------------

    // Update the quantity of "Widget" to 12.

    inventory.updateQuantity("Widget", 12);

    // -----------------------------

    // Using Operator Overloading for Direct Access

    // -----------------------------

    try {

        // Using the overloaded [] operator to access the first item.

        Item& firstItem = inventory[0];

        // Directly modify the quantity.

        firstItem.quantity = 100;

        cout << "\nAfter modifying the first item using operator[]:\n";

        print(firstItem);  // Print the updated first item.

    } catch (const out\_of\_range& ex) {

        cerr << ex.what() << "\n";

    }

    // -----------------------------

    // Printing Inventory Information

    // -----------------------------

    // Print the total number of items in the inventory.

    print(inventory.getItemCount());

    // Use the overloaded << operator to print the entire inventory.

    cout << "\nInventory (stack instance):\n" << inventory;

    cout << "\nInventory (dynamically allocated instance):\n" << \*inventoryPtr;

    // -----------------------------

    // Memory Cleanup

    // -----------------------------

    // Delete the dynamically allocated inventory to free memory.

    delete inventoryPtr;

    return 0;

}

**Step-by-Step Explanation:**

* **Creating Inventory Instances:**
  + Inventory inventory; creates an object on the stack (its memory is automatically managed).
  + Inventory\* inventoryPtr = new Inventory(); creates an object on the heap (we must manage it manually using delete later).
* **Adding Items:**
  + We add several items to both inventory objects using the addItem() function. When the inventory is full, the resize() function will automatically expand our array.
* **Updating an Item:**
  + The quantity of the "Widget" is updated from 10 to 12 using updateQuantity().
* **Using Operator Overloading (**operator[]**):**
  + The expression inventory[0] uses our overloaded operator to access the first item directly.
  + Changing firstItem.quantity demonstrates that the returned reference lets us modify the item in place.
* **Printing:**
  + We use the overloaded print() functions to display the total count and the details of one item.
  + The overloaded operator<< prints the full details of all items in the inventory.
* **Memory Cleanup:**
  + Before the program ends, we call delete inventoryPtr; to free the memory allocated with new.

**Final Thoughts**

* **Dynamic Memory Management:** The use of new and delete[] is central to managing memory in a dynamic array. The resize() function is critical because it allows the inventory to grow as more items are added.
* **Operator Overloading:** Operators like operator[] and operator<< make the class user-friendly. They allow you to access items with familiar array syntax and print the entire inventory easily.
* **Function Overloading:** The two versions of print() show how you can provide different functionalities based on the type of the argument passed.

By understanding each of these pieces, you can see how the program builds a robust inventory system step by step. Each part has a clear responsibility, and together they form a complete example of managing data with classes, arrays, operators, and memory in C++.