RuleOfThree

Joel Boynton

September 5, 2025

1 Problem Description

This is my understanding of the task at hand. We are given a main file and header file that are set up to emulate some type of game inventory system. The header acts as our implementation file in which we store all of our logic that will be used by our main file.

2 Solution Description

Since we are given the header file we can take a look at each of the tools that the main file will be using. By reading through the main file and its expected output we can get a sense of what each section is supposed to do. My process of determing how I will produce my solution comes from comparing the header and main files. My process is as follows: Read through the main file until I come across a function (constructors, operators, etc.). Once I find one, I take a look at what the expected output is supposed to be; Seeing if I can logically understand what the code is doing. Once this is done look for the given function in my header file and see what I am already given. From here I begin to code and confirm that my functions are working as intended.

3 Initial non-AI attempt to code the solution

My initial non-AI attempt went fairly well. I focused on each function independently of each other. Our Item class was going to function with a single item within our greater inventory. Here we have our Item constructor: this will pass a name and a quantity to our greater inventory that we will create. Alongside this we have to getters that can retrieve both the quantity and name of the item. These will be useful in developing the logic of the greater inventory.

Next we move on to our Inventory class. First we have our default constructor. This will create an array that can hold two item pointers. It is initialized to size zero and has a capacity of two. This effectively means you can hold up to two items before **resizing**. Next we have our copy constructor and assignment operator. Our copy constructor allows us to create a new inventory from an existing one. I use deepCopy in order new items that dont just copy the pointers. Once both inventories exist we use our assignment operator to assign our second, third, etc. inventory once it exists. Checks for self assignment and deletes old contents of the inventory. Then we use our

destructor to prevent memory leaks in case the inventory goes out of scope or is deleted.

The two most complicated areas here are the add and detete functions. addItem attempts to add a new item. This is done by looping over all items replaces the item located at **i**. removeItem does the opposite. It loops through each item and checks for the one we want to remove. Once it is removed the size is reduced by one. printInventory, just prints all our information to the terminal. We also have a few private helper methods that we use. resize, deepCopy, and freeMemory. resize will increase the capacity of our array, allowing room for more items then deletes the old array. deepCopy as mentioned before, creates a copy of every item (not the pointer). freeMemory loops through all items and deletes each one to prevent memory leaks.

I used AI to fix one compiling issue. but immediately noticed my output was not coming out as intended as everything was being deleted after adding an item from inventory. A few of my methods were deleting my inventory, but I quickly fixed them.

4 AI prompts used

I had trouble with my destructor as I kept getting an error when trying to compile claiming that I had an extra qualifcation. As I could not determine what this meant, I consulted AI presenting the line of code and asking why this might be happening. I fixed this easily as I just needed to remove part of the line. After this I could compile.

5 Code testing description

As we were given a main.cpp that provided a test run and an expected output. Running test was rather straightforward.

6 Code

```
#ifndef INVENTORY_H
#define INVENTORY_H

#include <iostream>
#include <string>
```

```
class Item {
public:
  Item(const std::string &name, int quantity)
      : name(name), quantity(quantity) {}

  std::string getName() const { return name; }

  int getQuantity() const { return quantity; }

private:
  std::string name;
  int quantity;
};

class Inventory {
public:
  Inventory() : items(new Item *[2]), size(0), capacity(2) {}
  Inventory(const Inventory &other) { deepCopy(other); }

  Inventory &operator=(const Inventory &other) {
    if (this != &other) {
      freeMemory();
      deepCopy(other);
    }
    return *this;
  }

  ~Inventory() { freeMemory(); }

  void addItem(const std::string &name, int quantity) {
    // Check if item already exists
    for (int i = 0; i < size; i++) {
      if(items[i]->getName() == name){
        delete items[i];
        items[i] = new Item(name, quantity);
        return;
      }

    }
  }
```

```
  void removeItem(const std::string &name) {
    for (int i = 0; i < size; i++) {
      if (items[i]->getName() == name) {
        delete items[i];
        for (int j = i; j < size - 1; j++) {
          items[j] = items[j + 1];
        }
        size--;
        return;
      }
    }
  }

  void printInventory() const {
    std::cout << "Inventory :";
    if (size == 0) {
      std::cout << " ( empty )\n";
    } else {
      std::cout << "\n";
      for (int i = 0; i < size; i++) {
        std::cout << "- " << items[i]->getName() << " ("
                  << items[i]->getQuantity() << ")\n";
      }
    }
  }

private:
  Item **items;
  int size;
  int capacity;

  void resize() {
    capacity *= 2;
    Item **newItems = new Item *[capacity];
    for (int i = 0; i < size; i++) {
      newItems[i] = new Item(items[i]->getName(), items[i]->getQuantity());
    }
    for (int i = 0; i < size; i ++){
      delete items[i];
```

```
    }
    delete[] items;
    items = newItems;
  }

  void deepCopy(const Inventory &other) {
    capacity = other.capacity;
    size = other.size;
    items = new Item *[capacity];
    for (int i = 0; i < size; i++) {
      items[i] =
          new Item(other.items[i]->getName(), other.items[i]->getQuantity());
    }
  }

  void freeMemory() {
    if(items){
      for(int i = 0; i < size; i++){
        delete items[i];
      }
      delete[] items;
      items = nullptr;
    }
    size = 0;
    capacity = 0;
  }
};

#endif


//TEST OUTPUT
[1] Newly created inventory
Inventory : ( empty )

[2] After adding 3 items
Inventory :
- Apples (10)
- Bananas (5)
- Carrots (12)
```

```
[3] After removing Bananas
Inventory :
- Apples (10)
- Carrots (12)

[4] Copy-constructed inventory (should match [3])
Inventory :
- Apples (10)
- Carrots (12)

[5] Original after removing Apples
Inventory :
- Carrots (12)

[6] Copy remains unchanged
Inventory :
- Apples (10)
- Carrots (12)

[7] Assigned-from-original inventory (should match [5])
Inventory :
- Carrots (12)
```