Homework 02

[2024-1] 데이터사이언스를 위한 컴퓨팅1 (001)

Due: 2024년 4월 30일

The purpose of this assignment is to familiarize oneself with the concepts of OOP, including inheritance, polymorphism, function overloading, overriding, class templates, and additionally, dynamic memory allocation.

1. Container[50pts]

This problem involves declaring objects related to containers capable of holding items and calculating the total capacity of the declared containers. Containers fundamentally receive their size in the constructor, determining their capacity accordingly. The Container class has the following hierarchy:
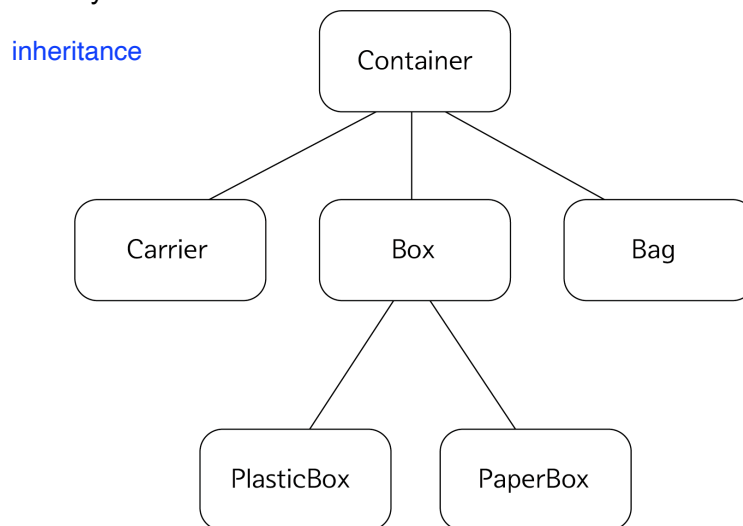

inheritance

Figure 1 Container Class Hierarchy

Instruction

Read the below instruction thoroughly and implement all classes and member functions in "container.hpp".

Role of each class:

pure virtual function

Container - This is the top-level **abstract class** serving as the parent for all containers. Container class and the derived classes receive a size as an input in float type during construction and has a structure where capacity is determined based on the size. Also, all containers are assigned independent codes based on their type and capacity(more details about codes will be explained on problem (d)).

like function overloading

- **Bag** - A derived class of Container where the capacity is size*3.
- **Box** - A derived class of Container where the capacity is size*1.
-       PaperBox - A derived class of Box where the capacity is size*1.3.
        PlasticBox - A derived class of Box where the capacity is size*1.1.
- **Carrier** - A child class of Container where the capacity is size*5. The key distinction

when constructed

from other containers is that it can attach up to two auxiliary containers of the Box class. The capacity of a Carrier is determined by adding the capacities of attached auxiliary containers to its original capacity, with the capacity of auxiliary containers being 1.1 times their original capacity.

The implementation proceeds sequentially through questions (a) to (e). To obtain full credit, all functions listed in each question must be implemented with the same names. Additional member functions can be defined as needed. Member variables can also be freely declared and used. **To receive full credits, functions and constructors from previous questions have to be implemented properly. In addition, if a compile error occurs in test case code, all test cases will be scored as 0 points. If no compile error occurs in main.cpp, there will be no compile errors in the test cases**. **Make sure to check it.**

(a) [10pts] Implement constructors and member variables for all classes based on Figure 1 and instructions. During the implementation of the constructor, you must read the instructions properly. The constructor takes a size as float. Implement the **float get_size()** function in the Container class to return the size.

(b) [10pts] Implement the **void set_capacity()** function for all classes using overriding. Also, implement the **float get_capacity()** function to return the capacity of the object in the Container class and make it available to all derived classes.

(c) [10 pts] Implement the **void add_aux(Box* aux)** function for the Carrier class to add a Box object to an empty auxiliary box slot. If both slots are full, it should perform no action. This function also updates the carrier's capacity according to the rules outlined in the instructions above. Additionally, implement the **Box* get_aux(int n)** function to return the pointer to the n-th auxiliary box (according to the order of assignment). n is either 1 or 2. This function should return nullptr if n is greater than the number of aux boxes attached to the current carrier.

(d) [10pts] Implement the **void set_container_code()** function in Container class. This function dynamically allocates a string object for the code and sets its value as follows. The code consists of the container type + capacity_code, where the capacity_code is "1" for capacities less than 100, "2" for 100 <= capacity < 200, and "3" for all others. The codes for each container type are: {"Bag":"B", "Carrier":"C", "PaperBox":"PaB", "PlasticBox":"PlB"}. For example, the code for a Bag with a capacity of 115 is "B2", and for a Carrier with a capacity of 500, it's "C3". Also, implement **string* get_container_code( )** function to return the pointer to the code.

(e) [5 pts] Implement proper destructors for each class. The destructors should be responsible for releasing memory allocated on the dynamic heap within the class.

(f) [5 pts] In this question, you don't need to implement something. Grading is conducted through cases that sum the total capacity of multiple containers. In the main.cpp file, there are functions void Episode1 and void Episode2. Test cases are structured in the same format as these two functions. The total capacity of all containers will be the comparison target. If you implemented (a)~(e) properly, you can get full credit from this question. Compare the results of running Episode1 and Episode2 with Figure 2 to verify if you have

implemented container.hpp correctly. Results of Episode1 and Episode2 do not affect grading, but matching their results with Figure 2 will guarantee it to receive credits during the test.

```
Episode 1:
Total capacity:700.2

Episode 2:
Total capacity:904.1
```

Figure 2 Result of main.cpp

2. Custom Map

This assignment involves implementing a map that is similar to std::map in the C++ standard library. A map typically consists of key-value pairs, and its construction allows for the specification of the types of keys and values through templates. Additionally, the template for the map takes std::less as the default type for Compare and std::allocator as the default type for Allocator. Figure 3 is the prototype of the actual C++ std::map.

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

Figure 3 Prototype of map

The map to be implemented in this assignment should be designed as a template that takes K as the data type for the key, V as the data type for the value, and Compare as a comparison class. In this assignment, you don't have to consider the Allocator type. std::less typically takes two data of the same type and compares which one is smaller. Below is an example usage of std::less. Additional explanations can be accessed through the link.

```
template <typename K, typename V, typename Compare = std::less<K>>
class map {
    private:
        // ....
        // other member variables
        // ....
        Compare comp;
        // ....
        // other member variables
        // ....
    public:
        // ....
        // other member functions
        // ....
        void example_function(K key1, K key2){
            if(comp(key1,key2)){
                cout << "Key1 is smaller than key2" << endl;
            }else if(comp(key2,key1)){
                cout << "Key2 is smaller than key1" << endl;
            }
        }
};
```

Figure 4 Example code of std::less

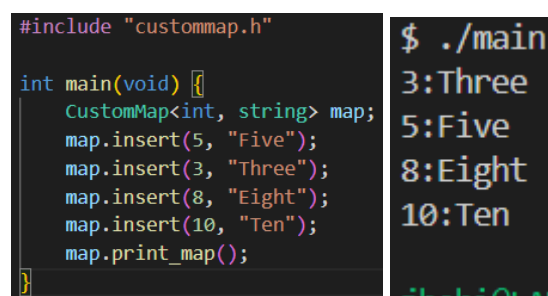The objective of this assignment is to implement a CustomMap that stores data in a Binary

Search Tree (BST). In this case, the binary search tree stores data based on the key, in accordance with the BST property, and the value is stored in the same TreeNode as a member variable. The goal is to implement operations such as insert, delete, and traverse, and operator[ ] using std::less for comparing keys.

Instruction

Read all explanations above and subproblems (a) through (e) below and implement them in "custommap.hpp". To receive full credit, you must declare all member functions and member variables as instructed in the subproblems. Additionally, declaring extra member functions and variables is permissible in the completion of the assignment. **If you don't implement the BST properly, time out errors can occur. Make sure to implement BST with O(logN) time complexity. Also, if you use std::map on your implementation, you will get 0 credits. Make sure to write your code not to contain "#include <map>" Not even a comment. Unless you will get 0 credits.**

(a) [10 pts] Define the TreeNode class using a class template. The class template should take two template arguments: the type of the keys, the type of the values. TreeNode should declare private member variables **K key**, **V value**, **TreeNode* left**, and **TreeNode* right**. The constructor should take K k and V v and assign them to **key** and **value** accordingly. Additionally, there should be getter and setter functions for each member variable. Name of each getter, setter function is not fixed, so you can name it freely. Similarly, define the CustomMap class using a class template. The class template should take three template arguments: the type of the keys, the type of the values, the compare class std::less<K> for comparison inside the map data structure(refer to Figure 4). CustomMap class should have TreeNode corresponding to the root of the BST, **TreeNode* root** and **Compare object** as member variables. The CustomMap constructor should not take any arguments. CustomMap class should have **TreeNode<K,V>* get_root( )** and **Compare get_compare( )** as a getter function.

(b) [10 pts] Write a **void insert(K key, V value)** function that inserts the key-value into the BST as a TreeNode, in accordance with the properties of a binary search tree. And also, write a function **void print_map()** that prints the key:value pairs stored within CustomMap in ascending order of keys. The output from running should be like figure 5. Make sure to print the "key:value\n" form exactly, without any space or other characters. Unless you won't get any credits. Grading based on test cases will rely on the output of the print_map() function. So even if the insert() function is implemented correctly, you will get no credits if the print_map() function is not implemented correctly.

```cpp
#include "custommap.h"

int main(void) {
    CustomMap<int, string> map;
    map.insert(5, "Five");
    map.insert(3, "Three");
    map.insert(8, "Eight");
    map.insert(10, "Ten");
    map.print_map();
}
```

```
$ ./main
3:Three
5:Five
8:Eight
10:Ten

jhshi@LAP
```

Figure 5 Result of insert and print_map

(c) [10 pts] Write a function **void deleteKey(K key)** that takes a key of type K as an argument and deletes the node corresponding to that key. If the input key does not exist, the delete function should perform nothing. Grading based on test cases will rely on the output of the print_map() function and insert function to construct the tree, so even if the deleteKey function is implemented correctly, you will get no credits if the print_map() and insert() function is not implemented correctly.

(d) [10 pts] Write a function **V get_value(K key)** that takes a key of type K as an argument and returns the corresponding value of type V. If the input key does not exist, the get_value function should return default value, V(). Grading based on test cases will rely on the insert function to construct the tree, so even if the get_value function is implemented correctly, you will get no credits if the insert() function is not implemented correctly.

(e) [10 pts] Overload the **[ ] operator** such that you can access a value using a key. When the key exists in the map, this operator returns the corresponding value. When the key does not exist, this operator inserts a new element with the key and the default value V( ). Also, this operator should allow for modifying the corresponding value directly. For example, by "map[3] = 4", this operator has to perform "adding a pair whose key is 3 and value is 4" if there is no key 3, and has to replace its value with 4 if the key exists. Grading based on test cases will rely on the insert function to construct the tree, so even if the operator[ ] is implemented correctly, you will get no credits if the insert() function is not implemented correctly.

3. Submission Instructions

- Implement the definition of the class and member functions together in "container.hpp" for Container, and similarly, implement the definition of the class and member functions together in "custommap.hpp" for Custom Map. You can modify other files for your own testing, but keep in mind that **only each hpp file will be graded using our test code.**

- Grading will be conducted using the C++11 standard. Our compilation process is as follows:

  o $ `g++ main.cpp -o main -std=c++11` (The main.cpp file includes the hpp files implemented in Container or Custom Map.)

- You should NOT share your code with other students. Any student found to have a high level of code similarity, as determined by our similarity detection process, may face severe penalties.

- If you submit late, grace days will be automatically deducted. You can utilize 5 grace days throughout the semester for homework submissions. Grace days are counted based on 24-hour increments from the original due time. For instance, if an assignment is due on 4/30 at 3:30pm, submitting it 30 minutes late will count as using one grace day, while submitting it on 5/1 at 9pm will count as using two grace

- For questions about this homework, use the "Homework 2" discussion forum on eTL.

- Failing to adhere to the submission guidelines may result in penalties applied without exception.