CSE 232 Spring 2014

Programming Project 10

This assignment is worth 60 points (6.0% of the course grade) and must be **completed and turned in before 11:59 on Monday, April 14th , 2014.**

The Problem

We are going to work on making our own container class using dynamically allocated memory. We are going to build a `Knapsack` class, which is also called a Bag or Multiset in computer science texts. You are then going to solve a Knapsack problem using your data structure.

Some Background

A `Knapsack` is best described by an example. Imagine you have some packages that you have to deliver in your delivery truck. Each package you have to deliver has two aspects:

- a **priority**
- a **weight**

You should deliver all your packages but the sum of the available packages exceeds the maximum weight you can carry in your truck. You have to make a decision, which packages to take. You should:

- take as many packages as you can carry in your truck such that you maximize the priority of the packages you deliver.

Said another way, you want to fill your truck such that

- the packages selected do not exceed the truck's maximum capacity
- the sum of the priorities of the selected packages is the maximum possible given the weight constraint.

This is often called the `Knapsack` problem. The `Knapsack` data structure is a container that can hold items of some type and has a fixed capacity (can hold a maximum weight). The problem is to fill the `Knapsack` to capacity while maximizing its priority.

We need to do three things to address this problem

- make a `Package` struct.
- make a `Knapsack` class
- write an algorithm to address the Knapsack problem.

In particular, *you cannot use an STL container inside of your Knapsack class*. Memory has to be dynamically allocated and deleted.

Interface, package.h

The `Package` is a good example of needing a `struct`, not a `class`. A `Package` exists to carry information on its weight and priority, that's about it.

- public data member `long weight_`
- public data member `long priority_`
- `Package(long weight, long priority);`
  - constructor.
- overloaded function `ostream &operator<<(ostream&, Package p),`
  - print a `Package.`

- doesn't have to be a friend since the members are public.

- `bool package_compare (const Package& lhs, const Package& rhs);`
  - this is a function, doesn't have to be a friend because all data members are public.
  - we compare two packages based on their ratio of `priority_/weight_`. Eventually we want to find those packages with the highest such ratio (most priority/weight) as those are the best packages to include in our `Knapsack`.
  - compares the `lhs Package` with the argument `rhs Package`, returning `true` if the `lhs Package` is larger (in `priority_/weight_` ratio), `false` otherwise.
  - in a `sort` of say a `vector<Package>`, you can use this function in the sort to order the vector.

Interface, knapsack.h file

- private data `Package* data_` the contents of the `Knapsack`
- private data `long weight_limit_,` the maximum weight the Knapsack can hold
- private data `long capacity_`, the size the underlying array (dynamically allocated) can hold before it needs to grow.
- private data `long size_`, the actual number of elements in the underlying array.
- *feel free to add anything else you think is important.*

- `Knapsack(long max)`. constructor, one argument.
  - the arg `max` is the maximum weight the Knapsack can take, no default
  - set the `capacity_` to 10, `size_` to 0, create the underlying array data_
  - again, you cannot use an STL data structure for this. You have to dynamically allocate memory for `data_` of your `Knapsack`.
- `bool add(Package p)`. member function, 1 argument of type `Package`
  - if, by adding the argument `Package` the `Knapsack` **exceeds** the capacity, then do not add `Package` to the contents of the `Knapsack,` return `false`.
  - if, by adding the argument `Package` the `Knapsack` **does not exceed** capacity, add the `Package` to the contents of the `Knapsack,` return `true`.
  - if the `Package` can be added to the `Knapsack` (by doing so the capacity of the `Knapsack` is not exceeded) **but the size of `contents_` is exceeded**, then you must:
    - dynamically allocate a `new_data` array that is twice the size of the previous `data_`
    - copy all the `Packages` from the old `data_` to the `new_data`
    - swap `data_` and `new_data`
    - delete `new_data`
    - add the `Package` to the contents of the `Knapsack`
- `bool empty()`. member function, no parameters
  - returns `true` if the `Knapsack` is empty, `false` otherwise.
- `long weight()`. Member function, no args
  - sum of the weight of the `Packages` the `Knapsack` currently holds
  - 0 if the `Knapsack` is empty
- `long priority()`. Member function, no args
  - sum of the priorities of `Packages` the `Knapsack` currently holds
  - 0 if the `Knapsack` is empty.
- `ostream& operator<<(ostream &out, const Knapsack &ks)`. This is a ***friend*** function (not a member). It prints the underlying `contents_` array and other elements of the class,

Algorithm, solveKS

```
void solve_KS(vector<Package>& vp, Knapsack& k);
```

This is a ***friend*** function. It adds elements to the Knapsack in a particular order as long as the weight limit is not violated. The following is not an optimal solution, but it is a decent solution

- sort the knapsack contents in order of `priority_/weight_`.
  - use the `package_compare` function in `package.h` in the sort
- take elements from the sorted vector, place them in the knapsack using the `add` method until you cannot take any more.

Deliverables

Turn in `package.h`, `package.cpp`, `knapsack.h` and `knapsack.cpp` files, using the handin program.
Save a copy to your H drive.

Notes

Notes will be updated.

Test

Implement your class and get it to run with the provided `main.cpp`. That main uses a file `packages.txt` to get things going.