CSE 232 Fall 2017

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 Wed, November 29th , 2017.** The change in the date is due to the Thanksgiving holidays the week before.

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, sort of, 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 it turns out that the sum of all 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:

- deliver as many of the most important/high-priority packages as you can. That is, maximize the sum of the priority of the packages you deliver.
- stay below the weight limit of truck. That is, the sum of the weight of the packages should be below the truck weight limit.

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 weight limit (can hold a maximum weight). The problem is to fill the `Knapsack` up to (but not over) its maximum weight 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, proj10_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. It represents the individual packages you are to deliver.

Data Members

- public data member `long weight_`
- public data member `long priority_`

Function Members

- `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 proj10_knapsack.h file

The Knapsack is the truck, the container of all the packages that you can take. It has a fixed weight limit set at construction. The sum of the weight of the packages that are placed in the Knapsack cannot exceed this limit. Since we cannot know how many packages we can place in the Knapsack before we exceed the weight limit, and because we are restricted from using STL containers, our underlying Knapsack will use an array that can grow as more Packages are added (up to the weight limit).

Data Members

- private data `Package* data_` the array contents of the `Knapsack`
- private data `long weight_limit_,` the maximum weight the Knapsack can hold
- private data `long capacity_`, the ***initial*** size (the number of `Packages`) the underlying array (dynamically allocated) can hold before it needs to grow. Default value 10 (in the header)
- private data `long size_`, the actual number of elements in the underlying array.

Function Members

- `Knapsack(long max)`. Constructor, one argument.
  - the arg `max` is the maximum weight the Knapsack can take, no default
  - 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`.

Getters

- `long capacity()const` . Member function, no args
  - return the present `capacity_` of the underlying array, the number of Packages the array could hold before having to grow.
- `long size() const` . Member function, no args
  - return the present `size_` , the number of Packages presently in the underlying array.
- `long weight_limit()const` . Member function, no args
  - return the present `weight_limit` that is set for this Knapsack instance.

Rule of Three Members

As we are working with dynamic memory, we need the following members.

- `Knapsack(Knapsack& )`. The copy constructor.
- `~Knapsack()` . The destructor.
- `operator=(Knapsack)` . Assignment operator

Other Members

- `bool add(Package p)`. member function, 1 argument of type `Package`
  - if, by adding the argument `Package` the `Knapsack` **exceeds** the `weight_limit_`, then do not add `Package` to the contents of the `Knapsack`, return `false`.
  - if, by adding the argument `Package` the `Knapsack` **does not exceed** `weight_limit_`, add the `Package` to the contents of the `Knapsack`, return `true`.
  - if the `Package` can be added to the `Knapsack` (by doing so the `weight_limit_`, of the `Knapsack` is not exceeded) **but the size of `data_` 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() const` . member function, no parameters
  - returns `true` if the `Knapsack` is empty, `false` otherwise.
- `long weight() const` . Member function, no args
  - sum of the weight of the `Packages` the `Knapsack` currently holds
  - 0 if the `Knapsack` is empty
- `long priority() const`. Member function, no args
  - sum of the priorities of `Packages` the `Knapsack` currently holds
  - 0 if the `Knapsack` is empty.

Friends

- `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, solve_KS

`void solve_KS(string fstring, Knapsack& k);`

- opens the file provided by fstring
  - if the file is not available, throw a runtime_error;
- if file opens, read each line which consists of a weight and a priority, space separated.

This is a ***friend*** function. It adds elements to the Knapsack from the opened file.
It does so in a particular order as long as the weight limit is not violated. To find the truly optimal arrangement of Packages is a bit beyond us at this point (see https://en.wikipedia.org/wiki/Knapsack_problem for a discussion). However, we can implement the following algorithm which is satisficing (does a good job) but is not guaranteed to be optimal.

- sort the knapsack `data_` array in order of `priority_/weight_`.
  - use the `package_compare` function in `package.h` in the sort

- take elements from the sorted array ***in order***, place them in the knapsack using the `add` method until you cannot take any more.

Deliverables

- Turn in `proj10_knapsack.cpp`, `proj10_package.cpp` in a `proj10` directory.
- Use Mimir for testing (as always). There will be hidden test cases.