COMP2401 - Assignment #3

(Due: **Monday, November 2nd, 2020 @ 6pm**)

In this assignment you will simulate a **Shopper** that goes to a grocery store and buys items. The **Shopper** will select **GroceryItems** and place them into an Array representing a shopping cart and then when buying the items, he/she will pack them into **GroceryBags** which will also use Arrays. From this assignment, you will gain experience in using **structs**, **pointers**, **malloc()** and **free()**.

Create a program to simulate items being purchased at the store. Call the program whatever you'd like. Grocery items will be placed into shopping carts and then packed into grocery bags. A shopping cart can contain at most **100** items. The maximum weight that a grocery bag can hold is **5kg** and the maximum number of items that can be placed in a bag is **25**. These should be fixed/constant limits within your program and hence defined as constants. Follow these instructions:

- Create a **type** called **GroceryItem** to represent an item that can be purchased from the grocery store. It should keep this the name of the item (e.g., "Cascade"), with no limit on name length, the cost of the item (e.g., $4.79), the weight of the item in Kilograms (e.g., 2.4 kg) and a flag of some sort indicating if this item needs to be refrigerated or frozen.

- Create a **type** called **Bag** that represents a bag that can hold grocery items. It should an array that will hold **pointers to** grocery items, a count of the current number of items in the bag and the total weight of all items currently in the bag.

- Create a **type** called **Cart** that represents a shopping cart with grocery items in it. A cart should keep an array that contains **pointers to** all the loose grocery items that the shopper is planning to purchase as well as the total number of items in the cart. It should also keep an array of packed grocery bags as well as the number of these bags in the cart. This array of bags MUST be dynamically-allocated, as well as the bags themselves. Therefore, the size of the array is unknown and should not be set to a fixed capacity.

- Create a function to add a grocery item to a shopping cart, taking a pointer to a grocery item and a pointer to a cart as parameters. Also, create functions to add an item to a bag and to remove an item from a bag. These functions should take pointers to a grocery item and a bag as parameters and they should all return **1** if successful and **0** otherwise.

- Create a procedure that takes a pointer to a grocery item and displays it in the format shown here. It should show the name, weight (with three decimal places) and properly formatted price.

```
SnackPack Pudding weighing 0.396kg with price $0.99
```

- Create a procedure that takes a pointer to a grocery bag as well as an integer representing the bag number and displays it in the format shown here. It should show the bag number and total weight (with three decimal places) of items in the bag. It should then display all the items in the bag, slightly indented with a '*' character to the left of the heaviest item in the bag. If there is more than one of those heaviest items in that bag, then each one should have a '*' beside it. You MUST also make use of the function that you just created to display a single grocery item.

```
BAG 6 (Total Weight = 4.068kg)
   SnackPack Pudding weighing 0.396kg with price $0.99
   SnackPack Pudding weighing 0.396kg with price $0.99
   Nabob Coffee weighing 0.326kg with price $3.99
  *Ocean Spray Cranberry Cocktail weighing 2.260kg with price $2.99
   Gold Seal Salmon weighing 0.213kg with price $1.99
   Heinz Beans Original weighing 0.477kg with price $0.79
```

- Create a procedure that takes a pointer to a shopping cart and displays it in the format shown here. It should show the unpacked items followed by the packed bags. You MUST make use of the two functions that you just wrote above.

```
Coca-Cola 12-pack weighing 5.112kg with price $3.49
Toilet Paper - 48 pack weighing 10.890kg with price $40.96
Toilet Paper - 48 pack weighing 10.890kg with price $40.96
Coca-Cola 12-pack weighing 5.112kg with price $3.49

BAG 1 (Total Weight = 3.302kg)
   Gold Seal Salmon weighing 0.213kg with price $1.99
   5-Alive Frozen Juice weighing 0.426kg with price $0.75
   5-Alive Frozen Juice weighing 0.426kg with price $0.75
  *Lean Ground Beef weighing 0.750kg with price $4.94
   Smart-Ones Frozen Entrees weighing 0.311kg with price $1.99
  *Lean Ground Beef weighing 0.750kg with price $4.94
   5-Alive Frozen Juice weighing 0.426kg with price $0.75

BAG 2 (Total Weight = 4.841kg)
   Ocean Spray Cranberry Cocktail weighing 2.260kg with price $2.99
   Smart-Ones Frozen Entrees weighing 0.311kg with price $1.99
  *Breyers Chocolate Icecream weighing 2.270kg with price $2.99

BAG 3 (Total Weight = 4.242kg)
   SnackPack Pudding weighing 0.396kg with price $0.99
   Heinz Beans Original weighing 0.477kg with price $0.79
   Smart-Ones Frozen Entrees weighing 0.311kg with price $1.99
  *Breyers Chocolate Icecream weighing 2.270kg with price $2.99
   Heinz Beans Original weighing 0.477kg with price $0.79
   Smart-Ones Frozen Entrees weighing 0.311kg with price $1.99
```

- Create a procedure that packs the items in a grocery cart into bags. It should dynamically-allocate a bag array and store it in the cart and then fill up the bags with items until they are full. After packing, the cart should contain any loose items that could not be packed and it should also contain a bunch of packed bags. The array of bags in the cart must be dynamically allocated to the exact number of bags needed. You may create a *temporary* "bigger" array of bags to solve this problem or you can first determine how many bags are needed. The packing algorithm that you must use is simple … just take the items from the cart in their sequential order and put them into a bag until the next item would cause the bag to exceed it's 25-item limit or 5kg weight capacity. In that case, you should begin packing a new bag. Once done, there should be no unpacked items in the cart except those that cannot be packed into a bag due to its weight.

- Create a procedure to remove the perishable items from the shopping cart. This procedure should go through all the packed bags and remove (by calling the function you wrote already) any items that are considered to be perishable. After this procedure has been called, there should be no perishable items left in the cart nor in any bags in the cart.

- The program should have a **main** function that creates exactly 12 grocery items with this data:

  ```
  GroceryItem  testItems[12];

  char *sampleItemNames[]   = {"Smart-Ones Frozen Entrees", "SnackPack Pudding",
  "Breyers Chocolate Icecream", "Nabob Coffee", "Gold Seal Salmon", "Ocean Spray
  Cranberry Cocktail", "Heinz Beans Original", "Lean Ground Beef", "5-Alive Frozen
  Juice", "Coca-Cola 12-pack", "Toilet Paper - 48 pack", "Crate of milk"};

  float sampleItemPrices[]  = {1.99, 0.99, 2.99, 3.99, 1.99, 2.99, 0.79, 4.94, 0.75,
  3.49, 40.96, 12.99};

  float sampleItemWeights[] = {0.311, 0.396, 2.27, 0.326, 0.213, 2.26, 0.477, 0.75,
  0.426, 5.112, 10.89f, 6.18f};

  float sampleItemPerish[]  = {1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1};
  ```

  Follow these steps in your main function:
  - populate the **testItems** array with the data above for initialize all the items.
  - create a shopping cart with 50 items randomly chosen from these 12 items. You will have many duplicate items. You must use your function that adds items to the cart.
  - call your function that displays the cart… indicate via text that it is *before packing*
  - call your function that packs the bags
  - call your function that displays the cart again… indicate via text that it is *after packing*
  - call your function that removes the perishables
  - call your function that displays the cart … indicate via text that it is *after removing perishables*

- In your code, make sure that a call to **valgrind** returns 0 errors and 0 memory leaks. Make sure that your code is efficient and that you are making good use of your written functions.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit all of your **c source code** files as a single **tar** file containing:

1. A **Readme** text file containing
   - your name and studentNumber
   - a list of source files submitted
   - any specific instructions for compiling and/or running your code

2. All of your **.c source** files and all other files needed for testing/running your programs.

3. Any output files required, if there are any.

The code **MUST** compile and run on the course VM.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not **written neatly with proper indentation and containing a reasonable number of comments.** See course notes for examples of what is proper indentation, writing style and reasonable commenting).