# CPSC 131, Data Structures – Spring 2020
# Grocery Item: Introduction & Review Homework

## Learning Goals:

- Become familiar with creating, compiling, running, and submitting programming assignments
- Demonstrate mastery of basic C++ skills, including
  - allocating and releasing dynamic memory
  - reading from standard input and writing to standard output
  - working with standard vectors
  - overloading insertion and extraction operators
- Demonstrate the ability to translate requirements into solutions
- Refresh your memory and normalize our point of departure. Depending on your background and how long ago you actively practiced programming in C++, some of this may be review and some may seem new to you

## Description:

In this assignment, you will play both the role of class designer and the role of class consumer. As class designer you will implement the provided class interface, and then then as class consumer you will instantiate objects of this class to solve a simple problem. The class itself has a few private attributes, and a multiparameter constructor. Objects of the class have the fundamental capability to insert, extract, and compare themselves. The problem being solved is simply to read several objects storing then dynamically, and after you've read them all print them in reverse order. Specifically, you are to implement:

1. **A class named GroceryItem**. You will reuse this class in future homework assignments, so effort you apply getting it right now will greatly benefit you later.
   a. Attributes
      i. Product Name – the name of the product (Ex: Heinz Tomato Ketchup - 2 Ct, Boston Market Spaghetti With Meatballs)
      ii. Brand Name – the product manufacture's brand name (Ex: Heinz, Boston Market)
      iii. UPC – a 14-digit Universal Product Code uniquely identifying this item (Ex: 00051600080015, 05017402006207). Code this as a string type, not an integral type.
      iv. Price – the cost of the item in US Dollars (Ex: 2.29, 1.19). Code this as type `double`.
   b. Construction
      i. Allow grocery items to be constructed with zero, one, two, three, or four arguments[1]. The first argument must be the product name, the second the brand name, the third the UPC, and the fourth the price.
      ii. Initialize each attribute with member initialization[2] and in the constructor's initialization list[3]. Do not set the attribute's value in the body of the constructor.
   c. Operations
      i. Set and retrieve each of the attributes. Name your overloaded functions upcCode, brandName, productName, and price. For example:

```
void        upcCode( const std::string & upcCode); //mutate
std::string upcCode() const;                       //query
```

[1] See Rational.hpp @ lines 91-92 and RationalArray.hpp @ lines 72-74 for constructors with multiple arguments, some with defaulted values
[2] See Rational.hpp @ lines 288-289 and RationalArray.hpp @ lines 211-216 for member initialization examples
[3] See Rational.hxx @ lines 45, 58 and RationalArray.hpp @ lines 58-59 for constructor's initialization list examples

ii. Overload the insertion and extraction operators[4]. For example, main() may read from standard input and write to standard output a GroceryItem object like this:

```
GroceryItem groceryItem;
std::cin  >> groceryItem; // extraction (reading)
std::cout << groceryItem; // insertion  (writing)
```

Insertion and extraction should be symmetrical. That is, you should be able to read what your write. Assume fields are separated by commas and string fields are always enclosed with double quotes. The first field must be the UPC, the second the brand, the third the product name, and the fourth the price. For example:

"00013000001038", "Heinz", "Heinz Tomato Ketchup - 2 Ct", 21.80

Incomplete items or items with errors should be ignored. Don't try to add and remove the quotes yourself. See and use std::quoted()[5])

iii. Overload the 6 relational operators[6]. For example, function main() may compare GroceryItem objects like this:

```
GroceryItem captainCrunch, cereal;
if( captainCrunch == cereal ) ...
if( captainCrunch  < cereal ) ...
```

Items A and B are equal if all attributes are equal (or within 0.0001 for floating point numbers). Grocery Items are to be sorted by UPC, brand name, product name, then price. For example, if A's and B's UPC are equal but A's brand name is less than B's brand name, then A is less than B.

d. Separate interface from implementation using header and source files. Implement all functions in the source (.cpp) file, not the header (.hpp) file. Header file only solutions will not be accepted.

2. **Function main()** to use the GroceryItem class above:
   a. Read a grocery item from standard input (std::cin) until end of file[7]. For each item read:
      i. Store the grocery item in a dynamically allocated object
      ii. Store the pointer to the grocery item in a standard vector
   b. After you have reached the end of file, write the grocery items to standard output (std::cout) in reverse order.
   c. Be sure to release the dynamically allocated objects before exiting the program

[4] See zyBooks section 1.17. Also see Rational.hpp @ lines 50-51 and Rational.hxx @ lines 155-156 for insertion and extraction overloading examples. Additional examples at operator overloading (scroll down to Stream extraction and insertion)

[5] Additional example at std::quoted and the "friendly" delimiters,

[6] See zyBook section 1.16. Alos see Rational.hpp @ lines 48, 361-377 and Rational.hxx @ lines 128-143 for relational operator overloading examples. Additional examples at operator overloading (scroll down to Relational operators)

[7] This program requires you to not open files. Simply write your program extracting data from std::cin. Enter Cntl-D (Linux) or Cntl-Z (windows) to indicate end-of-file. Better yet, create a text file with your input and then simply redirect input from that text file (see below). You know you have an incorrect solution if you have included <fstream> or call the ifstream::open function.)

# Reminders:

- The C++ using directive `using namespace std;` is **never allowed** in any header or source file in any deliverable products. Being new to C++, you may have used this in the past. If you haven't done so already, it's now time to shed this crutch and fully decorate your identifiers.
- Object Oriented programming suggests that objects know how to read and write themselves. Classes you write should overload the insertion and extraction operators.
- Object Oriented programming suggests that objects know how to compare themselves. Classes you write should overload the equality and inequality relational operators.
- Always initialize your class's class and instance attributes, either with member initialization, within the constructor's initialization list, or both. Avoid assigning initial values within the body of constructors.
- Use Build.sh on Tuffix to compile and link your program. There is nothing magic about Build.sh, all it does is save you (and me) from repeatedly typing the very long compile command and all the source files to compile. Using Build.sh is not required, but strongly encouraged. The grading tools use it, so if you want to know if you compile error and warning free (required to earn any credit) than you too should use it.
- Using std::system("pause") is not permitted. If you don't know what this is, good!
- Don't directly compare floating point numbers for equality. Instead of `x == y`, say `std::abs(x-y) < EPSILON`.
- Filenames are case sensitive, both in source code and in your OS file system. Windows doesn't care about filename case, but Linux does.
- You may redirect standard input from a text file, and you must redirect standard output to a text file named output.txt. Failure to include output.txt in your delivery indicates you were not able to execute your program and will be scored accordingly. A screenshot of your terminal window is not acceptable. See How to build and execute your programs. Also see How to use command redirection under Linux if you are unfamiliar with command line redirection.

# Deliverable Artifacts:

| Provided files | Files to deliver | Comments |
|---|---|---|
| GroceryItem.hpp | 1. GroceryItem.hpp | You should not modify this file. The grading process will overwrite whatever you deliver with the ones provided with this assignment. It is important that you deliver complete solutions, so don't omit this file in your delivery. |
| | 2. GroceryItem.cpp<br>3. main.cpp | Create these files as describe above |
| | 4. output.txt | Capture your program's output to this text file and include it in your delivery. Failure to deliver this file indicates you could not get your program to execute. |
| GroceryItemTests.cpp<br>CheckResults.hpp | | This source file contains code to test your GroceryItem class. When you're far enough along and ready to have your class tested, then place these files somewhere in your working directory alongside main.cpp and Build.sh will find it. Simply having these files in the directory will add it to your program and run the tests – you do not need to #include anything or call any functions. These tests will be added to your delivery and executed during the grading process. |
| sample_input.txt | | A sample set of data to get you started |
| sample_output.txt | | A sample of a working program's output. Your output may vary |