UMKC School of Computing and Engineering

Assignment #5 Due: 11/8

I. The assignment problem:

The UMKC's Bookstore wants you to build a store application for them. The application main goal is to keep track of the products purchased and to print a receipt at the end. As you know, the Bookstore sells different types of products. Nevertheless, we assume that our Bookstore sells T-shirts and office supplies only (for simplicity).

II. The application input:

The input to your application is a file, called products.txt. A sample file is given to you. This will contain all of the data needed to read in products bought.

The first data item is the product type. This will be "SHIRT" or "SUPPLIES". Based on the type, your application should read data of different formats:

- SHIRT: The next full line of the file is the description. This could be multiple words. The next line contains the size (which will be "S" through "3XL"). The next item is the quantity purchased, then the price per item *without* the dollar sign ($).
- SUPPLIES: The next full line of the file is the description. This could be multiple words. The next line contains the number of items inside of each set. After that is the quantity purchased, then the price per item.

III. The application output (you can skip to section IV then read this section):

The final output of your application should be a file called receipt.txt. This file will look similar to the following:

```
****************************************
*            UMKC BOOKSTORE            *
****************************************

Qty Description                 Total
--- --------------------------- --------
  3 M KC Royals shirt           $  44.97
  5 Envelopes (10 count)        $  24.95
  5 Envelopes (100 count)       $  49.95

Grand total: $119.87
Items sold:  3

* Thank you for shopping at our store! *
```

The store info at the top should consist of one line of 40 *s, one line of one *, spaces, the store's name, spaces, then one *. The final line is another string of 40 *s.

After this should be <u>one product per line:</u>

- The quantity should be printed right-aligned in a width of 3 spaces.
- The description should be in a set in a width of 28 spaces:
  - SHIRT: the size, a space, then the description
  - OFFICESUPPLIES: the description, a space, then a (, the count, a space, the word "count", then)
- The price should be printed with a $, then a width of 7 spaces. The price should always print out with two decimal places and have the decimal point also.
- There should be another blank line after all products, then print out "Grand total:" with the total price.
- The next line should print "Items sold:" with the number left-aligned with the price.
- The final line should be separated by a blank space and the happy message "* Thank you for shopping at our store! *".

IV. Application Classes

Your application must include four classes: Products, Shirt, OfficeSupplies, and Register. Following is a description of each of the classes.

❖ Product:

Product class is a generic product. The class will be abstract, and thus cannot be instantiated. The product class must include the following:

- Variables to store the price (double), quantity (integer), and description (string)
- A default constructor and a constructor that accepts price and quantity
- Setter functions to set the class's description, price, and quantity
- A pure virtual function, called print, that takes in a reference to an *ostream* and prints out one receipt line for this product. (more in the child elements)
- A virtual function, called calculateTotal, that will print out the total of how much the product will cost. For a generic product, it's just the quantity times the price.

❖ Shirt

The *Shirt* class is derived from the Product class. In addition to the inherited variables, Shirt class has a variable to store the shirt's size, called size. T-shirts come in various sizes, i. e., S – 3XL. The price is the same for all shirts, but the 2XL and 3XL shirts costs a dollar more. The Shirt class must include the following:

- A default constructor
- A function to set the size
- An overridden *print* function to print out how a Shirt item looks on a receipt. This also takes in a referenced *ostream*

- An overridden *calculateTotal* function to modify how each shirt's price is calculated. This is still price times quantity, but if it's 2XL or 3XL add a dollar to the cost.

❖ OfficeSupplies

The *OfficeSupplies* class is inherited from Product class. As you know, office supplies are often sold in sets. For example, 50-pack envelopes, 5-pack of pens, and 500-sheet reams of paper. You class must reflect this. In particular, your OfficeSupplies class must include the following:

- A variable to store the *count* of how many items of a certain product exist (e.g.: 5-pack of pens)
- A function to set the count
- An overridden print function to print out how a set of OfficeSupplies will print out. This also takes in a referenced *ostream*
- Since a 50-pack box still counts as one item, the cost is still quantity times price, and doesn't require redefining the *calculateTotal* function.

❖ Register

All the magic happens here. Once the register knows everything that has been purchased, it can print out a receipt for the customer. Each customer can purchase 50 products at most. The Register class will need:

- A default constructor
- An array of pointers to Products (more below). This can be a regular array.
- A variable to keep track of how many Products are stored in that array, you can call it *numProducts*
- A void function *addProduct* that will take in a pointer to a Product and add it to the array
- A void function *printReceipt* that will take in a reference to an *ostream* and print out the entire contents of the receipt to the output stream (more later).

To store polymorphic objects correctly, the register will need an array of pointers of Products. When you read in the data from the file, you will create a new type of Product based on what is read in. The pointer to that object will be passed into *addProduct*, and add the pointer to the array. After it's done, the array will point to all of the different types as appropriate.

To print out all of the objects correctly, they all have a required virtual print function. This function can be called on any Product type of object. The *printReceipt* function of this class can just simply loop through the array, then call the print function on each element of the array. The appropriate print function will be called on whatever type is in that array position. (For example, if Product[0] is a Shirt, it will call Shirt's print function.)

V. Header Guards

To prevent issues with having .h files being included more than once, you can include header guards to prevent the compiler from trying to include an object more than once. To do it, you need to use the pre-compiler directives #ifndef, #define, and #endif to set it up.

At the beginning of your .h file, after your #includes, type in the #ifndef ("if not defined") directive. Include a variable name that makes the file unique. (Ex: Product.h might have a variable PRODUCT_H.) After that, have a #define with the same variable name. Then write your class. At the end of the file, include the #endif.

Your classes should be declared in separate .h files and defined in separate .cpp files. The main function should be in its own .cpp file.

VI. Other information

- Your output file must match the one presented at the top.
- Since descriptions can have multiple words, but exists only on one line, you can use getline() on it. You may also find that, even though Product types are only one word, you may wish to use getline() also to reduce issues with new line characters not getting read in as often as needed.
- Write your classes in separate .h and .cpp files. You should turn in 9 files when you are done.
- If you use inheritance right, you won't be copying much code. Write Product first, then the others will be faster to implement.

Remember:

- **Build your code incrementally.**
- **Zip your project and upload it to Canvas before November 8th.**

~BestOfLuck()