# Catapult x SBG Homework

## The Problem

We’d like you to design a supermarket checkout that calculates the total price of a number of items.

Items are stored with their unit price.

Some items are eligible for discounts based on the following 2 deal types:

1. buy 3 identical items and pay for 2
2. buy 3 (in a set of items) and the cheapest is free

No other deal types need to be considered for the purposes of this exercise.

The output from the C++ program should be the customer receipt. This should show the price of each item bought and the grand total.

You may choose any sensible means of accepting input and producing output. You should submit to us the C++ source code and any associated documentation.

## Solution overview

The solution takes in an input from the user and then converts it to a list of items. The output is then stored and broken down into a vector of strings (referenced by a pointer to reduce memory consumption). The array is then iterated through and the value for each item is found using std::map – this then allows the program to find the prices and store them in a vector – where the index of the vector for item name corresponds to the index of the of the vector for the price.

These two vectors are then passed to a class representing the checkout system. The main checkout system class is an abstract class. This is because I have tried to base the code on the ability to scale up. A checkout system is common across many different supermarkets and other potential sectors – so there will be many shared values (e.g., Till ID). However, the deals, the types, accepted payments methods etc may all be different and therefore will be represented in derived classes. Given the context of the question – the virtual function (initialising the abstract class) is the calculation of discount. Furthermore, the methods for discount will be private functions in order to protect the data.

Then the calculation of price is represented by two functions – 3 for 2 and buy 3 get the cheapest free. The discounted price is then returned to the till and printed.

## Technical Breakdown

### User Input

The input is gathered using Cin. However, this considers spaces (whitespace and tabs) as a terminating character so in order to take in a whole comma separated string we needed to use getline() – the first parameter is the user input, the second is the output string and declare no delimiter (by leaving the last value blank).

These values are then sent to a string stream, separated into individual strings and pushed to a vector.

The member function getPrices is called.

### Map

The map is declared as a global variable and created in an initial function to simulate pulling data from a database – or linking the data and setting up a connection initially. Then the look up could be performed simultaneously.

Typically, when connecting to a database (in my experience anyway) a pointer to an object is returned and called when data is required. This could be added into the checkout class if required.

### Abstract Class

With class creation the use of a general checkoutSystem abstract class was thought to be most appropriate. This is because the base class would simulate a general checkout system which would be similar across the various sectors and chains of retail stores. However, there will then be many variants on internal workings such as deals, types of processing required, and different goods handled (clothes, food, office supplies). In this case the derived class was created as an instance of a supermarket checkout – which in this case is Asda. As of C++11 constructors could be inherited using ‘using’ – this way the common values (such as till ID, cash or card etc can be used. Therefore, I overloaded the default constructor to initialise the system on class creation.

The class is created using a unique pointer for two reasons:

1. To automatically manages memory deletion when out of scope.
2. It was mentioned in the interview if I knew had to use them (so I thought I would show you I could learn newer C++ techniques)

### Class variables

Here I thought I would give you a little insight into why I chose to add some extra complexity with regards to data members.

There are several vectors to store original items (pre savings), and then a list of items where savings were provided and then a final value. Any items that do not have a reference price are also returned for extra user output.

There are various debugs and return codes (-1) if any errors occur e.g., no items were input – an errors is raised and fed back to the user. This is important as error handling is often required in systems such as these.

Furthermore, with the member functions, the internal workings of the savings lists and prices were created with private access. This was done for several reasons:

1. A particular supermarket may want to protect the inner workings of till systems and deals.
2. You want to encapsulate and abstract the savings details and processes from the end user. Not only does this decouple the code from the user (making the UX cleaner) it also protects the deal information from leaks or being accessed by unauthorised individuals.

### Group of three identical items (3 for 2)

The general workings of the function are as follows:

1. The list is iterated through and the current item is checked and compared to the previous one.
2. If it is the same, a count variable is incremented, if 3 of the same item are found then the item and its price are pushed to two savings vectors for later use.
3. Indexes of items where savings are applied are stored in a temporary vector and later deleted providing an updated items list.
4. If not a new item is found and the process repeats till the end of the list

A tuple is then returned (containing all data related to savings and an updated item lists). This is done so that the deals cannot be applied to the same item twice. It is also due to the fact the data functions and variables are private.

### Group of three items function (buy 3 get the cheapest free)

The general workings of the function are as follows:

1. The number of times the list is divisible by three is checked (number of times savings needs to be applied).
2. For the number of savings that need to be applied the items list is iterated through and the item with the lowest price and its index is returned.
3. This item, its price and the index are pushed to the corresponding arrays.
4. The item is then deleted from the final items list (not necessary as such but if you want to add another deal afterwards this is important).

A tuple is returned for the same reasons as above (update savings list and data privacy for the final printing process).

### Receipt printed

I then used the lists and prices to print out a neat receipt to resemble a supermarket receipt (this was done using cout).

## Potential Future Developments

This was a really interesting challenge!

To improve this code further I would recommend the following changes:

1. Replace the cost comparison map to a database link(would expand the number of products available for input and reduce application size).
2. Multiple derived classes representing different branches/ chains of supermarket – each can have their own independent deals, prices and item functions.
3. Change internal private values to pointers – particularly when scaling up as not passing copies of variables to functions would improve compile time and run time performance. Given the amount of data here and present this isn’t an issue – however if numerous classes and thousands of prices were required then could affect the program performance.
4. Change Deals to a look up table of functions and pass outputs in a loop – more usable when scaling up.
5. Output the receipt to a UI – this way it could be output in an actual receipt format rather than via the terminal.