Unidays Discount Challenge 2019 Documentation

Submission by: Barry O'Connor
Email: me@barryoconnor.co.uk
Mobile: 07860497508

Compiling and Running the Program

GitHub Url: https://github.com/BarryOConnor/UnidaysDiscountChallenge

Firstly, simply clone the repository to your local computer.

Windows - Visual Studio

- If you are using the Visual Studio software suite, simply double click on the "UnidaysChallenge.sln" file.
- Once the project has loaded, you can compile the program by clicking on **Build->Compile** (Ctrl+F7)
- You can also compile and run the program by clicking on **Debug->Start Debugging** or pressing the F5 key.

Linux/Ubuntu - GCC

- Using the terminal (command line) navigate to the folder you cloned the GitHub repository to.
- Using the GCC compiler to compile the program, simply type

  ```
  g++ UnidaysChallenge.cpp UnidaysDiscountChallenge.cpp -Wall -o
  UnidaysDiscountChallenge
  ```

  This will compile the program, showing all warnings and save the compiled code to a file called UnidaysDiscountChallenge
- Once completed, you can run the compiled code by entering the following

  ```
  ./UnidaysDiscountChallenge
  ```

  Which will run the compiled code and display the results.

Design Considerations

Since this is a design challenge, I'm going to try and explain my thought processes while I was working on the challenge. I will include some more traditional documentation towards the end.

Understanding the Problem

Several things were clear from the initial writeup of the challenge. These gave me some ideas about what was required and served as a foundation for tackling the challenge.

The initialisation of the class would need to pass all the Pricing Rules to the newly created object for initialisation in some format.

This immediately suggested using a struct since each of the pricing rules is comprised of several elements. I also felt that there could potentially be an unknown amount of pricing rules, so a vector would allow me to push each pricing rule to the vector instead of using an array, which would have needed to be initialised with a size. This would allow for as many Pricing Rules to be input as necessary.

With that in mind, I decided to create a structure called PricingRule and to add those to a vector, which I would then pass to the newly-created class to initialise it with those pricing rules

The class would need to pass back some form of structure or object since the suggested code uses those notations.

The example given clearly requires the values for Total and DeliveryCharge would be expected to be properties of the returned type from the CalculateTotalPrice method. This prompted me to create a CheckoutTotals struct which would be used by the class as the return type.

The class would need to use a basic shopping basket functionality which would keep a count of the number of each item

I felt that this would make sense to be a struct, with the basket being a vector. This would be able to handle the user adding varying amounts of items by dynamically resizing. Using a struct which contained the item name and the current count for that item also made sense.

Challenges Faced Inputting Pricing Rules

I felt that the way the pricing rules were listed could cause issues if this process wasn't made very clear. There are many ways to refer to "buy one get one free" so using a string is cumbersome and prone to errors, if you had to type in hundreds of these, it would likely be difficult to keep consistency.

To get around this, I created an enum (enumerated list) with names for all the discounts. These can be changed if the client prefers other names and can be extended to include any other offers they may require. This standardises the discount names and can help programmers since Most GUI development environments will auto-suggest while you type saving keystrokes.

It was clear that some of the Pricing Rules required more information than others. Buy One Get One Free will always equate to 2 items costing the same as 1, however, 2 for £10 is not only the name of the deal but also the final price for both items. To get around this, I decided to add an extra item to the rules. The fourth item, discountPrice, would contain the price of the discount in cases where it wasn't easy to work out the discounted price from the discount itself. This would also allow different items to be part of the “Two For –” discount while having different prices.

Using the above amendments, the given pricing rules would now look like the following. Since some discounts don’t need a discountPrice, I also made it optional for those discount types that don’t require it when inputting to save on a few keystrokes.

Item	unitPrice	discountType	discountPrice
A	£8.00	NONE	N/A
B	£12.00	TWO_FOR	£20.00
C	£4.00	THREE_FOR	£10.00
D	£7.00	BOGOF	N/A
E	$5.00	THREE_FOR_TWO	N/A

Further thoughts on PricingRules

While working on pricing rules, I wanted to enter some basic sanity checks that would trigger at the development stage. I included a check for the fourth parameter for those discount types that needed it and added some basic checking of prices that would print a message if any of the prices entered were negative numbers. I felt that store owners would not be happy selling items at a negative price.

Adding items to the Basket

When adding items, if an item type exists already, it makes sense to just increment the counter for the item. If an item does not exist, then the item must be added and the corresponding count set to 1. With this in mind, I would need to check within the vector for an existing item. To this end, I researched and came up with a lambda function to do this, which I use several times in the code.

I also thought that adding an item to the basket which had no corresponding Pricing Rule should create a warning. Such items would have no price and would potentially be included in a sale without being charged for.

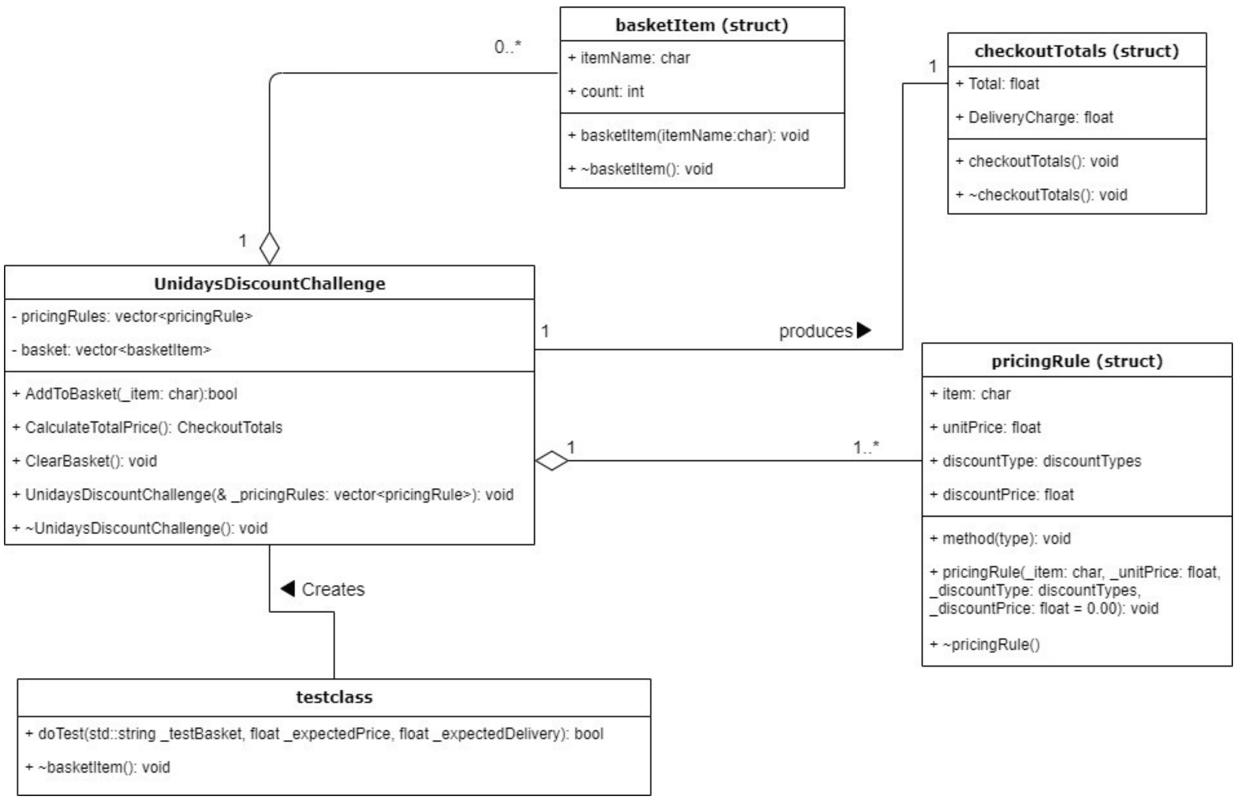
Calculating totals and using defaults

Calculating the discounts was relatively simple using the unitPrice in combination with the discountPrice. While doing this, I also decided to include a fallback default option for any items in the basket which don’t match any of the discount types.

I felt that the store owner would prefer items to be sold at the unitPrice as a default. This would mean that any errors in inputting the Pricing Rules would at least result in the item being sold at no discount. Logically, customers would be much more likely to notify the store if a discount wasn’t applied than they might be if the wrong discount was applied (to their benefit).

Technical Documentation

Class Diagram


basketItem (struct)
+ itemName: char
+ count: int
+ basketItem(itemName:char): void
+ ~basketItem(): void
checkoutTotals (struct)
+ Total: float
+ DeliveryCharge: float
+ checkoutTotals(): void
+ ~checkoutTotals(): void
UnidaysDiscountChallenge
- pricingRules: vector<pricingRule>
- basket: vector<basketItem>
+ AddToBasket(_item: char):bool
+ CalculateTotalPrice(): CheckoutTotals
+ ClearBasket(): void
+ UnidaysDiscountChallenge(& _pricingRules: vector<pricingRule>): void
+ ~UnidaysDiscountChallenge(): void
pricingRule (struct)
+ item: char
+ unitPrice: float
+ discountType: discountTypes
+ discountPrice: float
+ method(type): void
+ pricingRule(_item: char, _unitPrice: float, _discountType: discountTypes, _discountPrice: float = 0.00): void
+ ~pricingRule()
testclass
+ doTest(std::string _testBasket, float _expectedPrice, float _expectedDelivery): bool
+ ~basketItem(): void
0..*
1
1
1
produces ▶
1
1..*
◀ Creates

The main class is the UnidaysDiscountChallenge class which uses the other structs to function. A struct is essentially a class which consists of properties and methods which are all public.

The pricingRule struct is used to store the Pricing Rules defined at the start of the program. It has a relationship with the UnidaysDiscountChallenge class such that each UnidaysDiscountChallenge object can contain one or more pricingRules and each pricingRule can be allocated to 1 UnidaysDiscountChallenge object.

The basketItem struct is used to store items a user puts in their virtual basket. It has a relationship with the UnidaysDiscountChallenge class such that the UnidaysDiscountChallenge object can contain many basketItem and each basketItem can be allocated to 1 UnidaysDiscountChallenge object.

Testing

I created an additional class which I used for testing purposes. The specification only called for items to be added to the basket one at a time, which is cumbersome to do for large combinations of items. I wanted to be able to quickly run multiple tests for different combinations of products and to be able to have one method which would allow me to pass in expected result values and report on whether they were met.

For the sake of this challenge, I included the test class into the same header and cpp file as the UnidaysDiscountChallenge class for ease of reading and use. Normally this would not be included and would exist as a separate class of its own which could be removed from the project for release builds.

The following test cases were followed during development. The original test cases set by the challenge have all been included as well as 4 test cases I created to test for specific cases where the Price Rules had the potential to allow a user to buy something for a negative price or where the Price Rules were lacking an expected fourth parameter. The following data was used and can be followed for verification.

Additional Test Cases

Test Scenario	Test Steps	Test Data	Expected Results	Actual Results
unitPrice < 0.00	Add a negative unitPrice to a Pricing Rule and run	-4.00	Warning message shown	As expected
discountPrice < 0.00	Add a negative discountPrice to a Pricing Rule and run	-16.00	Warning message shown	As expected
discountPrice missing for TWO_FOR or THREE_FOR discountType	Remove a discountPrice from the corresponding rule	`appPricingRules.push_back({ 'B', 12.00, TWO_FOR });`	Warning message shown	As expected

Item added to the basket with no pricing rule	Add an item which has no testing rule	`appPricingRules.push_back({ 'A', 8.00, NONE });`
		 `example.AddToBasket('B');`
			Warning message shown	As expected

Catch an uncoded discount type	Add a new Pricing Rule with an “UNCODED” discount type.
	 Add an item to the basket with that pricing rule.
		`appPricingRules.push_back({ 'F', 11.00, UNCODED });`
		 `example.AddToBasket('F');`
			Behaves as if no discount were coded	As expected

Original Test Cases

Basket Contents	Total Price	Delivery Charge	Actual Results
None	£0.00	£0.00	As expected
A	£8.00	£7.00	As expected
B	£12.00	£7.00	As expected
C	£4.00	£7.00	As expected
D	£7.00	£7.00	As expected
E	£5.00	£7.00	As expected
BB	£20.00	£7.00	As expected
BBB	£32.00	£7.00	As expected
BBBB	£40.00	£7.00	As expected
CCC	£10.00	£7.00	As expected
CCCC	£14.00	£7.00	As expected
DD	£7.00	£7.00	As expected
DDD	£14.00	£7.00	As expected
EE	£10.00	£7.00	As expected
EEE	£10.00	£7.00	As expected
EEEE	£15.00	£7.00	As expected
DDDDDDDDDDDDDD	£49.00	£7.00	As expected
BBBBCCC	£50.00	£0.00	As expected
ABBCCCDDEE	£55.00	£0.00	As expected
EDCBAEDCBC	£55.00	£0.00	As expected