C++ code kata: Week #1 `VowelsCounter` Solution

Hello 👋

If you have still not attempted, I advise you to write at least a few solutions before moving forward with this solution document. For others here are the solutions 🎁 for you.

Ground Solutions

Let us start with legacy C/ C++ solutions. These are straight forward and easy to implement if you have always followed mostly towards language construct only and not any library. So these solutions go like this 🚀

Solution 1

```
01: #include <string.h>
02: int vowel_counter_010(const char * pPhrase){
03:     int nLen = strlen(pPhrase);
04:     int nVowels = 0;
05: 
06:     for (int i = 0; i < nLen; ++i) {
07:         if (pPhrase[i] == 'a' || pPhrase[i] == 'A' ||
08:             pPhrase[i] == 'e' || pPhrase[i] == 'E' ||
09:             pPhrase[i] == 'i' || pPhrase[i] == 'I' ||
10:             pPhrase[i] == 'o' || pPhrase[i] == 'O' ||
11:             pPhrase[i] == 'u' || pPhrase[i] == 'U')
12:             ++nVowels;
13:     }
14:     return nVowels;
15: }
16: 
```

Solution 1: Commentry

This is one of the naive approaches to count the occurrence of an item *in our case the vowels* within a container *in our case string* `pPhrase`. In this version of our solution, we have initially counted total characters in container *See line 03* using **strlen**.

This makes our enumeration *See line 06* a counter which can actually be a source of bugs. Also, the `if statement` *See line 07 to 12* needs to check for the character in both lower case and upper case which is quite tedious code as you have to do two lookups and comparison.

This makes our `if statement` time consuming and again can be a source of bugs. This can't be recommended as an ideal way for writing `VowelCounter` solution.

If you see in the last example we have written solution that is quite plain, difficult to read and understand. Our next solution tries to do things in other ways. Here we go 🚀

Solution 2

```
01: int vowel_counter_020(const char * pPhrase){
02:     int nVowels = 0;
03:
04:     for (int i = 0; pPhrase[i]; ++i){
05:         if ((pPhrase[i] | 32) == 'a' ||
06:             (pPhrase[i] | 32) == 'e' ||
07:             (pPhrase[i] | 32) == 'i' ||
08:             (pPhrase[i] | 32) == 'o' ||
09:             (pPhrase[i] | 32) == 'u')
10:             ++nVowels;
11:     }
12:     return nVowels;
13: }
```

Solution 2: Commentry

The only difference in this solution from the last is that we are using a trick to convert ith character to lowercase *See line 05 to 10* using biwise OR.

Though it looks smart it can be a source of bugs and relies on knowledge of Bitwise Operators. This solution is not the ideal one as it is just a tricky code otherwise it is exactly as last one. Our next solution is not the elegant one but it is just trying to reduce the stress of `for the statement`. Let's see how here we go 🚀

Solution 3

```
01: #include <string>
02: using namespace std;
03: int vowel_counter_030(const char * pPhrase){
04:     int nVowels = 0;
05:
06:     for (auto i: string(pPhrase))
07:         switch (i | 32){
08:             case 'a':
09:             case 'e':
10:             case 'i':
11:             case 'o':
12:             case 'u':
13:                 ++nVowels;
14:         }
15:     return nVowels;
16: }
```

Solution 3: Commentry

If you are paying attention *See line 06* we are making use of **colletions loop**.

This change is actually converting our approach for iterating from **counter loop** to actually **collection iterator **. It means that we are now getting the actual item from the collection instead of its index number.

Also, we are using `auto`, making compiler to automatically deduce the type using initializer. Though now a great example you should make it a habit to use `auto` while working with STL and generics. If this would have been a Unicode character string `auto` should be the choice to avoid certain bugs.

As we are basically doing a pattern searching within a container, we can make use of regular expressions to help us out. Let's see how.

Here we go 🚀

Solution 4

```
01: #include <string>
02: #include <regex>
03:
04: int vowel_counter_040(const char * pPhrase)
05: {
06:     std::string objPhrase = (std::string(pPhrase));
07:     std::smatch objMatch;
08:     std::regex expr("[aeiouAEIOU]");
09:     int nVowels = 0;
10:     while (std::regex_search(objPhrase, objMatch, expr)){
11:         objPhrase = objMatch.suffix().str();
12:         ++nVowels;
13:     }
14:     return nVowels;
15: }
16:
```

Solution 4: Commentry

This time we are using regular expression to search an occurrence within the passed argument. Let's discuss how we did it.

We are using a smatch i.e. match_results _See line 07_object to initiate the regular expression matching that basically holds the character matching results. The actual matching is done by regex_match *See line 10*. It returns `true` if there is a match otherwise it returns false.

The regular expression to be matched is passed to regex object, that basically stores the regular expression as a character sequence.

On every successful match, we are updating the search phrase with `objPhrase` with remaining of string to be matched *See line 11*. Rest of code is self-explanatory.

Our next solution is again regex based but converts the search phrase to lowercase before moving forward.
Lets see how 🚀

Solution 5

```
01: #include <string>
02: #include <regex>
03: using namespace std;
04:
05: int vowel_counter_041(const char * pPhrase)
06: {
07:     string objPhrase = pPhrase;
08:     smatch objMatch;
09:     regex expr("[aeiou]");
10:     int nVowels = 0;
11:
12:     std::transform(objPhrase.begin(), objPhrase.end(), objPhrase.begin(),
::tolower);
13:     while (std::regex_search(objPhrase, objMatch, expr)){
14:         objPhrase = objMatch.suffix().str();
15:         ++nVowels;
16:     }
17:     return nVowels;
18: }
19:
```

Solution 5: Commentry

This solution is an augmentation of our last solution in which we have done 1 change. We have implemented **std::transform** which works over a range e.g. array or string and applies an algorithm *See line no 12*. In our case, it is converting the whole passed argument `objPhrase` to lowercase. This will help us to perform a match for lower case alphabets only and reducing comparisons and match time.

Our last solution puts another set of C++ standard library algorithms and data structure to work that can give us result without much fuss. Here we go 🚀

Solution 6

```
01: #include <string>
02: #include <regex>
03: #include <functional>
04: using namespace std;
05:
06:
07: int vowel_counter_050(const char * pPhrase)
08: {
09:     string objPhrase = pPhrase;
10:     return std::count_if(objPhrase.begin(), objPhrase.end(), [](char ch)
{return strchr("aeiou", ch | 32); });
11: }
12:
```

Solution 6: Commentry

This solution makes use of count_if algorithm *See line no 10*, that checks within a range (*`objPhrase`*) in our case_ and returns the count of elements for whom the predicate is true.

We are making use of lambda expression that is basically a quick functor. There are a few more changes that you can do to this solution that I am leaving for you to figure out and implement.

The test script for the solution can be **found here**

Hope you have liked this problem and its solutions. See you next Monday. Till then

Happy programming (Days)😉