C++ code kata: Week #4 `Mexican Wave` 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

The legacy C/ C++ solution is really easy to implement and goes like this 🚀

Solution 1

```cpp
#include <ctype.h>
using namespace std;

char **mexican_wave(const char *pPhrase)
{

    int nIndex = 0;
    int nLen = strlen(pPhrase);

    char **pMexicanWave = new char *[nLen ? nLen - 1 : nLen + 1] { const_cast<char *>("") };

    while (pPhrase[nIndex])
    {
        pMexicanWave[nIndex] = new char[nLen]();
        strcpy(pMexicanWave[nIndex], pPhrase);
        pMexicanWave[nIndex][nIndex] = toupper(pMexicanWave[nIndex][nIndex]);
        ++nIndex;
    }

    return pMexicanWave;
}
```

Solution 1: Commentry

We have designed our function `mexican_wave` to take one argument which is a **pointer** to characters. Since pointer will not change, we have kept it as **constant**. Our function returns a pointer to pointer to a character, which is a standard way to return **array of character pointer** in C / C++. Recall we cannot return the size of array this way.

To start with, we have allocated an array of character pointers `pMexicanWave` using `new` operator equal to the number of required **Mexican Waves**. Next, our algorithm iterates the argument array until we meet the NULL character. *See line number 12.*

In each iteration, we have allocated an array of characters and copied source array `phrase` into solution array i.e. `pMexicanWave`. *See line number 14 to 15*. The best part of this solution is that using the `nIndex` we can access source array, destination array as well location of character that we have to change to uppercase in a given `Wave`. In line 16 we are changing the target character from lowercase to uppercase.

At last, we have returned the solution which is an array of character pointers. There is, however, one weak point in this solution i.e we have no control over the lifetime of allocated memory i.e. there are sure memory leaks if not taken care of. Although we can tackle this problem using the smart pointers quite easily as we have done in `Week 3`, I have left it as an exercise to you.

If you see in the last example we have written solution that is quite plain, difficult to read as well as welcoming sure memory leaks. Whenever possible we should replace such solution with better alternatives. Out next solution shows us how to do it. Here we go 🚀

Solution 2

```
01: #include <vector>
02: #include <algorithm>
03: using namespace std;
04:
05: std::vector<std::string> mexican_wave(const char *pPhrase)
06: {
07:     std::string szPhrase = pPhrase;
08:     std::vector<std::string> mexicanWave;
09:     for (int x = 0; x < szPhrase.size(); ++x)
10:     {
11:         std::string szDestinationPhrase = pPhrase;
12:         szDestinationPhrase.at(x) = std::toupper(szDestinationPhrase.at(x));
13:         mexicanWave.push_back(szDestinationPhrase);
14:     }
15:     return mexicanWave;
16: }
```

Solution 2: Commentry

In this version of our solution, we replace our pointer to pointer to character `char **pMexicanWave` with a vector`std::vector<std::string> mexicanWave`. C++ STL's **vector** class is a sequential container that can grow as we add new elements. Thus we can use it, instead of static arrays or dynamically allocated arrays (as we have done in our last solution). *See line 08*. Our code is now much cleaner, easier to read and bingo! all

memory leaks are things of past. Further, this is a **string** vector, i.e. each entry in this vector is a string. Reason for this is simple, strings are much easier to maintain than character pointers. *See line 11 to 12*.
This problem is best tackled using alphabet index, and we have done that again in this solution using **at()**, although one could have used **operator[]**. Next, we have stored the newly created string into a vector.The **push_back** method of **vector** class will make vector to grow as new elements will come. 😃

We have used **toupper** to get uppercase alphabet. *See line 11 to 13*. As evident from code itself not only it is more readable but now there is less room for bugs.

If you have paid attention to the code then our last two solutions rely on the alphabet index to create a `Mexican Wave`. Obviously, that is a natural solution to the problem at hand. Although I won't recommend, let us see a solution that uses alphabet matching to get the desired result. Although it is much costlier. Let's see how here we go 🚀

Solution 3

```
01: #include <vector>
02: #include <algorithm>
03: using namespace std;
04:
05: std::vector<std::string> mexican_wave(std::string szPhrase)
06: {
07:     std::vector<std::string> mexicanWave;
08:     for (char target : szPhrase)
09:     {
10:
11:         std::string szDestinationPhrase(szPhrase);
12:
13:         std::transform(szPhrase.begin(), szPhrase.end(),
szDestinationPhrase.begin(), [target](char &ch) { return target == ch ?
std::toupper(ch) : ch; });
14:         mexicanWave.push_back(szDestinationPhrase);
15:     }
16:     return mexicanWave;
17: }
```

Solution 3: Commentry

There are a few things worth looking here. First, we have modified the signature of the function to accept a string instead of a char pointer. This is a perfectly acceptable and much better way to accept the character array intended to be a string. *See line 05*. C++ **STL** is a great library with many algorithms that we can make use of to further reduce the space for bugs 🐛 and at the same time achieve desired results and make our code more readable. AS we are dealing with a **container** we can safely use **range based loop**. *See line 08*.
Next, we are making a new string out of `szPhrase` and using **std::transform** to convert particular alphabet to uppercase. Nw this is very costly as we are comparing values sequentially instead of directly changing

alphabet using an index. Remember both vector and string are sequences i.e you can use the index operator to access value directly. Rest of code is self-explanatory.

Let's now move on to Level Up.

Level 1 Solution

Our last ground solution has already given up a huge jump to reach level 1. Although quite self-explanatory let's discuss it also. Here we go 🚀

Solution 1

```
File: lvl1_010.cpp
01: #include <string>
02: #include <vector>
03: #include <algorithm>
04: using namespace std;
05:
06: std::vector<std::string> mexican_wave_lvl1_010(std::string szPhrase)
07: {
08:     std::vector<std::string> mexicanWave;
09:     std::string::iterator start = szPhrase.begin();
10:     while (start != szPhrase.end())
11:     {
12:         if (!isspace(*start))
13:         {
14:             std::string szDestinationPhrase(szPhrase);
15:             szDestinationPhrase[start - szPhrase.begin()] =
toupper(szDestinationPhrase[start - szPhrase.begin()]);
16:             mexicanWave.push_back(szDestinationPhrase);
17:         }
18:         ++start;
19:     }
20:     return mexicanWave;
21: }
```

Solution 1: Commentry

In this solution, we have to deal with space characters other than alphabets. We have taken a lesson from our last solutions and moved back to indexing instead of comparing alphabets as it is not that good. Well as all containers in C++ STL use **iterators** to enumerate the collection we have taken a **bidirectonal iterator** for our passed argument `szString`. An iterator can move to next element in container till **end()**. **See line 09 to 10**. Now iterators, when negated, return the distance in terms of the number of locations. We have just this fact to our advantage for calculating the index. Though this is also costly this is done intentionally to show you operations on iterators. **See line 15**. Rest of code is self-explanatory except line 18 that I am leaving to your imagination.

Other Comments

To test all the ground up and level solutions I have created two helper functions (actually an overloaded one 😁). They basically compare the values until the matching lasts. For a successful match, in the first one should come zero, in the second one it should come equal length.

```
bool assertEqual(const char **pIterable1, const char **pIterable2, int nLen)
{
    /*Write normalization and comparison code here*/
    /*Return comparison result*/
    do
    {
        if (strcmp(pIterable1[nLen - 1], pIterable2[nLen - 1]))
            break;
    } while (--nLen);

    return !nLen;
}
bool assertEqual(vector<string> szIterable1, const char **pIterable2)
{
    int nIndex = 0;

    for (auto szString : szIterable1)
        if (szString.compare(pIterable2[nIndex++]))
            break;
    int nC = szIterable1.size();
    return nIndex == szIterable1.size();
}
```

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)😉