C++ code kata: Week #3 `Ordered Counter` 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

To start with, if you are coming from plain C/ C++ 🌐 then most possibly you may have written your **First** solution like this 🚀

Solution 1

```
#include <string.h>
#include <new>
using namespace std;


int * orderedCounter(const char * const pPhrase){
#define kALPHABET 52U
    int szAlphabets [kALPHABET] = {0};
    int nIndex;
    int nFrequency;

    for (nIndex = 0; pPhrase[nIndex]; ++nIndex)
        if (pPhrase[nIndex]>='A' && pPhrase[nIndex]<='Z')
            szAlphabets [pPhrase[nIndex] - 'A'] ++;
        else if (pPhrase[nIndex]>='a' && pPhrase[nIndex]<='z')
            szAlphabets [pPhrase[nIndex] - 'a' + 26] ++;

    nFrequency = 0;
    for (nIndex = 0 ; nIndex < kALPHABET; ++nIndex)
        if(szAlphabets [nIndex])
            ++nFrequency;

    int *szFreq = new int[nFrequency + 1]();
    nIndex = 0;
    for (nFrequency = nIndex = 0 ; nIndex < strlen(pPhrase); ++nIndex)
        if ((pPhrase[nIndex]>='A' && pPhrase[nIndex]<='Z') || (pPhrase[nIndex]>='a' && pPhrase[nIndex]<='z'))
            if (szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A':
```

```
pPhrase[nIndex] - 'a' + 26])
28:             {
29:                 szFreq [nFrequency++] = szAlphabets[pPhrase[nIndex]<'a'?
pPhrase[nIndex] - 'A': pPhrase[nIndex] - 'a' + 26];
30:                 szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A':
pPhrase[nIndex] - 'a' + 26] = 0;
31:             }
32:     return szFreq;
33: }
```

Solution 1: Commentry

We have designed our function `orderedCounter` to take one argument which is a **pointer** to characters. Since nor pointer or values will change both are kept as **constant**. Our function returns a pointer to an integer, which is a standard way to return arrays in C / C++. Recall we cannot return the size of array this way.

To start with, our algorithm first iterates the argument and stores the count of an individual alphabet (both capital and small) in an array. *See line number 08 to 16*.

Next, we added the total times all alphabets are coming in the argument. *See line number 18 to 21*. For this, we only have enumerated the array that we have prepared earlier i.e. `szAlphabets`.

Further, we allocated an array using new operator having size exactly equal to summation last calculated + 1 byte. *See line 23*.We didn't used and static array because we need to create a data structure that stores count exactly in the same order as the alphabets are coming in the phrase. We have to note here also that 0 < summation < length of the phrase. So, if you are tempted to create an array equal to the length of phrase statically, it will not only introduce bugs but memory also, in cases when alphabets are repeating mostly. e.g. "recess".

At last, we have copied the frequency of individual alphabet from `szAlphabets` into `szFreq` making use of `pPhrase` as the lookup key in `szAlphabets`.

If you see the last example we have written certain comparison codes that are not only difficult to read but can also be a welcome point for bugs if opt written carefully. Whenever possible we should replace such statements with better alternatives. Out next solution shows us how to do it. Here we go 🚀

Solution 2

```
01: #include <string.h>
02: #include <ctype.h>
03: #include <new>
04: using namespace std;
05:
06: int * orderedCounter(const char * const pPhrase){
07: #define kALPHABET 52U
08:     int szAlphabets [kALPHABET] = {0};
09:     int nIndex;
10:     int nFrequency;
11:
12:     for (nIndex = 0; pPhrase[nIndex]; ++nIndex)
13:         if (isupper(pPhrase[nIndex]))
14:             szAlphabets [pPhrase[nIndex] - 'A'] ++;
15:         else if (islower(pPhrase[nIndex]))
16:             szAlphabets [pPhrase[nIndex] - 'a' + 26] ++;
17:
18:     nFrequency = 0;
19:     for (nIndex = 0 ; nIndex < 52; ++nIndex)
20:         if(szAlphabets [nIndex])
21:             ++nFrequency;
22:
23:     int *szFreq = new int[nFrequency + 1]();
24:     nIndex = 0;
25:     for (nFrequency = nIndex = 0 ; nIndex < strlen(pPhrase); ++nIndex)
26:         if (isalpha(pPhrase[nIndex]))
27:             if (szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A':
pPhrase[nIndex] - 'a' + 26]){
28:                 szFreq [nFrequency++] = szAlphabets[pPhrase[nIndex]<'a'?
pPhrase[nIndex] - 'A': pPhrase[nIndex] - 'a' + 26];
29:                 szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A':
pPhrase[nIndex] - 'a' + 26] = 0;
30:             }
31:
32:     return szFreq;
33: }
```

Solution 2: Commentry

In this version of our solutions, we have replaced certain character comparison statements with library functions. C/ C++ library had a rich set of **character** manipulation function that every developer should pay attention to when working with characters.

We have used **isupper** and **islower** to verify if character is uppercase alphabet or lowercase. *See line 13 and 15*. Further, we have used **isalpha** to verify if the character is an alphabet or not. 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 have a major bug enjoying the time there. It is the **memory leakage** 👺 that is happening in form of our returned array of alphabet frequency. *See line 32 in the last two solutions*_.

Memory management has always been a headache 😟 for many C/ C++ developers in past. They say it is a place where even angels 😇 fear to tread. Well, the good news is that Modern C++ 💪 has introduced many new technologies to handle this in form of smart pointers. Let's see how here we go 🚀

Solution 3

```
01: #include <new>
02: #include <memory>
03: using namespace std;
04: #include <string.h>
05: #include <ctype.h>
06:
07: unique_ptr<int[]> orderedCounter(const char * const pPhrase){
08: #define kALPHABET 52U
09:     int szAlphabets [kALPHABET] = {0};
10:     int nIndex;
11:     int nFrequency;
12:
13:     for (nIndex = 0; pPhrase[nIndex]; ++nIndex)
14:         if (isupper(pPhrase[nIndex]))
15:             szAlphabets [pPhrase[nIndex] - 'A'] ++;
16:         else if (islower(pPhrase[nIndex]))
17:             szAlphabets [pPhrase[nIndex] - 'a' + 26] ++;
18:
19:     nFrequency = 0;
20:     for (auto a : szAlphabets)
21:         if(a) ++nFrequency;
22:
23:     unique_ptr<int[]> freq (new int[nFrequency + 1]());
24:     nIndex = 0;
25:     for (nFrequency = nIndex = 0 ; nIndex < strlen(pPhrase); ++nIndex)
26:         if (isalpha(pPhrase[nIndex]))
27:             if (szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A':
pPhrase[nIndex] - 'a' + 26]){
28:                 freq [nFrequency++] = szAlphabets[pPhrase[nIndex]<'a'?
pPhrase[nIndex] - 'A': pPhrase[nIndex] - 'a' + 26];
29:                 szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A':
pPhrase[nIndex] - 'a' + 26] = 0;
30:             }
31:
32:     return freq;
33: }
```

Solution 3: Commentry

What we have done here is that instead of just allocating raw bytes and using it we have now passed the allocated memory to **unique_ptr**. It is one of the new smart pointer class for automatic handling of allocated memory. Bingo! the memory will we reclaimed once goes out of scope. The only difference from the last examples is the usage of the smart pointer instead of a raw pointer. *See line 23*.

C++ STL library is a rich resource to many constructs and containers that we can make use of to further reduce the space for bugs 🐛 to grow and at the same time make our code more readable. Let's see how we do it. Here we go 🚀

Solution 4

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

vector<int> orderedCounter(const char * const pPhrase){
#define kALPHABET 52U
    int szAlphabets [kALPHABET] = {0};
    int nIndex;

    for (auto value: string(pPhrase))
        if (isupper(value)) szAlphabets[value - 'A'] ++;
        else if (islower(value)) szAlphabets [value - 'a' + 26] ++;

    vector<int> freq;
    nIndex = 0;
    for (nIndex = 0 ; nIndex < strlen(pPhrase); ++nIndex)
        if (isalpha(pPhrase[nIndex]))
            if (szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A': pPhrase[nIndex] - 'a' + 26]){
                freq.push_back(szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A': pPhrase[nIndex] - 'a' + 26]);
                szAlphabets[pPhrase[nIndex]<'a'? pPhrase[nIndex] - 'A': pPhrase[nIndex] - 'a' + 26] = 0;
            }

    if(freq.empty()) freq.push_back(0);
    return freq;
```

Solution 4: Commentry

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 solutions), in places where do not make a prior estimate of elements in the sequential container. As you can see we are not making use of total frequency counting and memory allocation as we have done in the last examples.

The **push_back** method of **vector** class will make vector to grow as new elements will come. 😃

If you are paying attention to our solutions from the last few weeks we are making use of **colletions loop**. We have again used it in Solution 3 and 4.

Till now in solutions, we have used an array `szAlphabets` to store the frequency of individual alphabets. When we know that many of locations in this array will remain 0 since not all alphabets from a to z are going to be part of the argument. In our next example, we have come up with a plan to get rid of this array entirely. Let see how. Here we go 🚀

Solution 5

```cpp
#include <new>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

#include <string.h>
#include <ctype.h>

class VALUETYPE {
    private:
        int m_nFreq;
        char m_cLetter;

    public:
        VALUETYPE (int nFreq, char cLetter):m_nFreq(nFreq), m_cLetter(cLetter) {}
        int frequency()const{return m_nFreq;}
        char letter()const{return m_cLetter;}
        void incrementFrequency(){m_nFreq += 1;}
        void setLetter(char cLetter){m_cLetter += cLetter;}
        bool operator == (char const &cLetter){ return m_cLetter == cLetter;}

};

vector<int> orderedCounter(const char * const pPhrase){

    vector<VALUETYPE> freqHolder;

    for (char value: string(pPhrase)){

        vector<VALUETYPE>::iterator itr = std::find(freqHolder.begin(), freqHolder.end(),value);
        if(itr != freqHolder.end())
            itr->incrementFrequency();
        else if(isalpha(value))
           freqHolder.push_back(VALUETYPE(1, value));
    }

    vector<int> freq;
    for (auto value: freqHolder)
        freq.push_back(value.frequency());

    if(freq.empty()) freq.push_back(0);
    return freq;
}
```

Solution 5: Commentry

We have written a class VALUETYPE that will store each character occurring in argument pPhrase as well as its frequency. *see line 10 to 23*. We are also using a vector now *see line 27* to store the occurrence of each alphabet in the passed argument. Now instead of having an array of 52 integers (as we doing in last solutions), we are storing an exact number of character as per frequency. This has made our code more robust and resilient to bugs 🐛 Now we are also using **find** algorithm to lookup an alphabet in vector and change its occurrence. This code is much cleaner, better, faster and easier to read. See how we have **overloaded = operator**😎 that is going to help us in implementing **find** operation.

STL is a great boon to build structures of our concepts. It is like a **Lego Brick** that you can use to create forms and shapes as you wish. Let see how. Here we go 🚀

Solution 6

```
01: #include <new>
02: #include <vector>
03: #include <map>
04: #include <algorithm>
05: #include <string>
06: using namespace std;
07: #include <string.h>
08: #include <ctype.h>
09:
10: vector<int> orderedCounter(const char * const pPhrase){
11:     map<char, int> freqHolder;
12:
13:     for (char value: string(pPhrase)){
14:         map<char, int>::iterator itr = freqHolder.find(value);
15:         if(itr != freqHolder.end())
16:             itr->second++;
17:         else if(isalpha(value))
18:            freqHolder.insert(std::make_pair(value, 1));
19:     }
20:
21:     vector<int> freq;
22:     int nIndex = 0;
23:
24:     while (pPhrase[nIndex]){
25:         map<char, int>::iterator itr = freqHolder.find(pPhrase[nIndex++]);
26:         if(itr != freqHolder.end()){
27:             freq.push_back(itr->second);
28:             freqHolder.erase(itr);
29:         }
30:     }
31:
32:     if(freq.empty()) freq.push_back(0);
33:     return freq;
34: }
35:
```

Solution 6: Commentry

In this last of ground solution we have replaced the VALUETYPE with **STL**'s very own **map**. If you have read about it by now, then **map** is a container that works on the logic of key-value pair.

For our example, we have taken alphabet as key and frequency as value. Since every element in the map is a pair, we have taken help of **make_pair** to store the values inside the map.

Using map has helped us to get rid of VALUETYPE class, had written incorrectly, it would have introduced bugs to our code. Also, our code is much easier to maintain and has become a pleasureful read.

Let's now move on to Level Up.

Level 1 Solutions

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

```
#include <new>
#include <vector>
#include <map>
#include <algorithm>
#include <string>
using namespace std;
#include <string.h>
#include <ctype.h>

vector<pair<char, int>> orderedCounter_level1_010(const char * const pPhrase){
#define kALPHABET 52U
    map<char, int> freqHolder;

    for (char value: string(pPhrase)){
        map<char, int>::iterator itr = freqHolder.find(value);
        if(itr != freqHolder.end())
            itr->second++;
        else if(isalpha(value))
            freqHolder.insert(std::make_pair(value, 1));
    }

    vector<pair<char, int>> freq;
    int nIndex = 0;

    while (pPhrase[nIndex]){
        map<char, int>::iterator itr = freqHolder.find(pPhrase[nIndex++]);
        if(itr != freqHolder.end()){
            freq.push_back(make_pair(itr->first, itr->second));
            freqHolder.erase(itr);
        }
    }
    if(freq.empty()) freq.push_back(make_pair(NULL, 0));
    return freq;
}
```

Solution 1: Commentry

In this solution, we have to return not only the frequency but also the alphabet in order of occurrence. That means we need a sequential container with heterogeneous values in a pair. The solution is quite easy to write as we have used our beloved vector to store the frequency. The catch is that instead of storing only frequency we have stored both alphabet and frequency as a pair. Again we went shopping at STL and came with **std::pair** to do the job. Rest of code is the same except for return type that I guess you would have understood by now.

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 till matching lasts and return the count of letters remaining. For a successful match, it should come zero.

```
bool assertIterableEqual (int *pIterable1, int *pIterable2, int nCount){

    while (*pIterable1++ == *pIterable2++)
        --nCount;
    return !nCount;
}

bool assertIterableEqual (std::pair<char,int> *pIterable1, std::pair<char,int>
*pIterable2, int nCount){
    while(pIterable1->first == pIterable2->first && pIterable1->second ==
pIterable2->second){
        ++pIterable1;
        ++pIterable2;
        --nCount;
    }

    return !nCount;
}
```

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)😉