C++ code kata: Week #5 `Anagram` 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 language only construct and not any library. So these solutions go like this 🚀

Solution 1

```
bool is_anagram(const char *pFirst, const char *pSecond)
{
    #define ALPHABETS 26

    int nFirst[ALPHABETS] = {0};
    int nSecond[ALPHABETS] = {0};
    int nIndex = 0;

    while (pFirst[nIndex])
        ++nFirst[(pFirst[nIndex++]|32) - 'a'];

    nIndex = 0;
    while (pSecond[nIndex])
        ++nSecond[(pSecond[nIndex++]|32) - 'a'];

    nIndex = 0;
    while (nIndex < ALPHABETS)
    {
        if (nFirst[nIndex] != nSecond[nIndex])
            break;
        ++nIndex;
    }
    return nIndex == ALPHABETS;
}
```

Solution 1: Commentry

This is one of the weakest approaches to discover anagram among strings. Here, we have designed our function `is_anagram` to take two strings arguments that we need to test. Strings have a usual way of passing which is as a **pointer** to characters. Since pointer will not change, we have kept it as **constant**. Our function returns **boolean** value indicating if strings are anagram or not.

To start with, our algorithm first iterates the arguments and stores the count of an individual alphabet (converting alphabets to small temporarily) into two arrays i.e. `int nFirst[ALPHABETS]` and `int nSecond[ALPHABETS]`. *See line number 05 to 14*. Just try to think why both these arrays have size `ALPHABETS`.

Next, our algorithm iterates both the arrays `nFirst[ALPHABETS]` and `nSecond[ALPHABETS]` from 0 to 26 until we meet first non-matching locations. *See line number 16 to 22*. If both the passed arguments i.e. `const char *pFirst` and `const char *pSecond` are anagram then `nIndex` will cover all the locations.

Our return value is a boolean expression that validates if `nIndex` iterated successfully or not.
The integer arrays that we have defined in last solution *See line number 05 and 06* are not required we can do without it. Let us see in our next 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

```c
01: #include <string.h>
02:
03: bool is_anagram(const char *pFirst, const char *pSecond){
04:     int nFirstIndex = 0;
05:
06:     if (strlen(pFirst) != strlen(pSecond))
07:         return false;
08:
09:     while( pFirst[nFirstIndex])
10:     {
11:         int nSecondIndex = 0;
12:         while(pSecond[nSecondIndex])
13:         {
14:             if((pFirst[nFirstIndex]|32) == (pSecond[nSecondIndex]|32))
15:                 break;
16:             ++nSecondIndex;
17:         }
18:         if(!pSecond[nSecondIndex])
19:             break;
20:         ++nFirstIndex;
21:     }
22:     return !pFirst[nFirstIndex];
23: }
```

Solution 2: Commentry

In this version of our solution, we have initially checked using **strlen** if both strings are of equal length or not. If not, we can safely return false. *See line 06 and 07*.

Next, we have matched each alphabet of `pFirst` with each alphabet of `pSecond`. *See line 09 and 21*. Since the passed strings may contain both lower and upper case characters we have normalized the individual alphabets to lower case temporarily at the time of comparison. *See line 14 and 15*. If we have iterated the `pSecond` completely that means there is nonmatching alphabet in `pFirst`. We have checked this fact and stopped matching further since it is clear that passed arguments are not an anagram. *See line 18 to 22*.

This solution is very costly as the calculation time for anagram matching is very high. Our next solution is not the elegant one but it is just trying to reduce the stress of expression writing. Let's see how here we go

---

Solution 3

```
01: #include <ctype.h>
02:
03: bool is_anagram(const char *pFirst, const char *pSecond)
04: {
05:     int nFirstIndex = 0;
06:
07:     while( pFirst[nFirstIndex])
08:     {
09:         int nSecondIndex = 0;
10:         while(pSecond[nSecondIndex])
11:         {
12:             if(tolower(pFirst[nFirstIndex]) == tolower(pSecond[nSecondIndex]))
13:                 break;
14:             ++nSecondIndex;
15:         }
16:         if(!pSecond[nSecondIndex])
17:             break;
18:         ++nFirstIndex;
19:     }
20:     return pFirst[nFirstIndex] == pSecond[nFirstIndex];
21: }
```

Solution 3: Commentry

There are a few things happening in this solution that reduces our code. First is the usage of **tolower**, for normalization of alphabets as we discussed. It checks for equality of elements as we did earlier. *See line 07 to 19*. To make sure that both strings have ended i.e. they are the equal length it doesn't use strlen() but checks character at `nFirstIndex`. *See line no 20*. As loop statement of line 07 breaks on '\0' NULL character, thus index character at both `pFirst`and `pSecond`should be the same indicating they are equal length and terminated.

It is easier to write an Anagram solution if we change our approach of comparing the alphabets. Our next written solutions make use of this approach. In all the coming solutions we will:-

1. Normalize both strings to lower case.
2. Sort both the string in **lexiographic** order so that we can get the result in a single iteration.

Making these changes will make our solutions clear to understand and much faster to execute. Here we go 🚀

Solution 4

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

void sort(char *pSequence)
{
    std::transform(pSequence, pSequence + strlen(pSequence), pSequence, ::tolower);
    int nLen = strlen(pSequence);

    for (int i = 0; i < nLen - 1; ++i)
        for (int j = i + 1; j < nLen; ++j)
            if (pSequence[i] > pSequence[j])
                swap(pSequence[i], pSequence[j]);
}

bool is_anagram(const char *pFirst, const char *pSecond)
{
    char *pSortedFirst = new char[strlen(pFirst)]();
    char *pSortedSecond = new char[strlen(pSecond)]();

    strcpy(pSortedFirst, pFirst);
    strcpy(pSortedSecond, pSecond);

    sort(pSortedFirst);
    sort(pSortedSecond);

    int nIndex = 0;
    while (pSortedFirst[nIndex] && pSortedFirst[nIndex] == pSortedSecond[nIndex])
        ++nIndex;

    bool bIsMatch = pSortedFirst[nIndex] == pSortedSecond[nIndex];
    delete[] pSortedFirst;
    delete[] pSortedSecond;

    return bIsMatch;
}
```

```
38:
```

Solution 4: Commentry

In this solution, we have first allocated memory to store the strings we need to sort, calculating their size using the **strlen** function. This is necessary since passed arguments are read-only memory i.e. **string literals**. *See line 19 to 24*. Next, we passed these strings to sort function. Before sorting we need to normalize the strings for that we have used **transform** that we are seeing for past few weeks. Next, we have used **selection sort** for sorting the strings. Also, we are using **std:: swap** to exchange the alphabets between locations instead of using a temporary variable. *See line 06 to 15*.

Next, we have implemented a loop that holds or executes till individual alphabets in two strings i.e. `pSortedFirst` and `pSortedSecond` are equal and current value in the first string isn't '\0' NULL.

Once out of the loop we compare current alphabet in both strings to be the same (it should be the NULL character). Next, we have deleted the allocated memory and return the result.

In the next solution, we will try to go towards automatic memory management and better sorting alternative instead of trying to implement it manually since sorting and memory management aren't the central theme of our solution. Here we go 🚀

Solution 5

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

bool is_anagram(const char *pFirst, const char *pSecond)
{
    std::unique_ptr<char[]> first(new char[strlen(pFirst)]);
    std::unique_ptr<char[]> second(new char[strlen(pSecond)]);

    strcpy(first.get(), pFirst);
    strcpy(second.get(), pSecond);

    std::transform(first.get(), first.get() + strlen(first.get()), first.get(), ::tolower);
    std::transform(second.get(), second.get() + strlen(second.get()), second.get(), ::tolower);
    std::sort(first.get(), first.get() + strlen(first.get()));
    std::sort(second.get(), second.get() + strlen(second.get()));

    return !strcmp(first.get(), second.get());
}
```

Solution 5: Commentry

This solution is an augmentation of our last solution in which we have done two changes. We have implemented **std::unique_ptr** for automatic memory management which is a better way than using the raw pointers and relieves our solution from memory leaks.

Also, we have replaced our version of sorting algorithm with **std:: sort**. This function takes starting and ending iterator and a comparator to compare individual values. As we have not provided a comparator it uses default comparator **less**.

This time as our strings are sorted we have used `strcmp()` to compare two strings for equality.

Our next solution tries to make use of more Standard C++ data structure to remove all nonrequired computation and focus completely over anagram calculation. Here we go 🚀

Solution 6

```
01: #include <algorithm>
02: #include <memory>
03: #include <string>
04: using namespace std;
05:
06: bool is_anagram(std::string first, std::string second)
07: {
08:
09:     std::transform(first.begin(), first.end(), first.begin(), ::tolower);
10:     std::transform(second.begin(), second.end(), second.begin(), ::tolower);
11:
12:     std::sort(first.begin(), first.end());
13:     std::sort(second.begin(), second.end());
14:
15:     return !first.compare(second);
16: }
```

Solution 6: Commentry

The only difference this solution is having than the last solution is the usage of **std:: string** for argument capture. *See line no 06*. This not only relieves the solution from manual iterator management, *See line no 09 to 13*, but also makes string comparison and result providing very easy. *See line no. 16*.

If we can ignore the transformation of strings to lowercase than this solution fits perfectly into our requirement.

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)😉