TOPIC - STRINGS

Introduction

Strings are actually one-dimensional arrays of characters terminated by a null character '\0'. Declaring a string is as simple as declaring a one dimensional array. Below is the basic syntax for declaring a string in C programming language.

char str_name[size];

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello." **char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};**

If you follow the rule of array initialization then you can write the above statement as follows: **char greeting[] = "Hello";**

Functions supported in C language that manipulate null-terminated strings:

strcpy(s1, s2)	Copies string s2 into string s1.
strcat(s1, s2)	Concatenates string s2 onto the end of string s1.
strlen(s1)	Returns the length of string s1.
strcmp(s1, s2)	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strchr(s1, ch)	Returns a pointer to the first occurrence of character ch in string s1.

strstr(s1, s2)	Returns a pointer to the first occurrence of string s2 in string s1.

Demonstration of some functions is given below:

```
#include <stdio.h>
#include <string.h>
int main () {
        char str1[12] = "Hello";
        char str2[12] = "World";
        char str3[12];
        int  len ;
        /* copy str1 into str3 */
        strcpy(str3, str1);
        printf("strcpy( str3, str1) :  %s\n", str3 );
        /* concatenates str1 and str2 */
        strcat( str1, str2);
        printf("strcat( str1, str2):   %s\n", str1 );
        /* total length of str1 after concatenation */
        len = strlen(str1);
        printf("strlen(str1) :  %d\n", len );
        return 0;
}
```

Output:

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

C++ provides following two types of string representations:

- The C-style character string (as discussed above)
- The string class type introduced with Standard C++

String Class in C++

String is an object of **std::string** class that represents sequence of characters.

Some Important Functions supported by String Class:

getline()	Used to store a stream of characters as entered by the user.
push_back()	Used to input a character at the end of the string.
pop_back()	Used to delete the last character from the string.
resize()	Changes the size of string, the size can be increased or decreased.
length()	This function finds the length of the string.
clear()	Clears all the characters from the string and string becomes empty.
empty()	Tests whether the string is empty and returns a Boolean value.

There are many other functions which can be found here. Demonstration of some is given below:

```
#include<iostream>
#include<string> // for string class
using namespace std;
int main()
{
```

```
    // Declaring string
    string str;

    // Taking string input using getline(), ("Hello Student" in this case)
    getline(cin,str);
    cout << "The initial string is : " << str << endl;

    // Inserts 's' at the end
    str.push_back('s');
    cout << "The string after push_back operation is : " << str << endl;

    // Deletes 's' (character from end) from the end
    str.pop_back();
    cout << "The string after pop_back operation is : " << str << endl;

    // Resizing string using resize()
    str.resize(10);
    cout << "The string after resize operation is : " << str << endl;

    cout<<"The length of the string is :"<< str.length()<<endl;

    str.clear();

    // Checking if string is empty
    (str.empty()==1)?
        cout << "String is empty" << endl:
        cout << "String is not empty" << endl;

    return 0;
}
```

Output:

The initial string is : Hello Student

The string after push_back operation is : Hello Students

The string after pop_back operation is : Hello Student
The string after resize operation is : Hello Stud
The length of the string is : 10
String is empty

Substring of a string

A substring is a contiguous sequence of characters within a string. In C++, std::substr() is a predefined function used for finding the substring of a string.

This function takes two values **pos** and **len** as an argument and returns a newly constructed string object with its value initialized to a copy of a substring of this object. Copying of string starts from *pos* and done till *pos+len* means **[pos, pos+len)**.

```
string substr (size_t pos, size_t len) const;
```

```
#include <string.h>
#include <iostream>
using namespace std;
int main()
{
        string s1 = "Hello Student";
        // Copy five characters ofs1 (starting
        // from position 3)
        string r = s1.substr(3, 5);
        cout << "Substring is: " << r;
        return 0;
}
```

Output:
Substring is: lo St

Practice Problem:

- https://www.geeksforgeeks.org/program-print-substrings-given-string/

- https://www.geeksforgeeks.org/sum-of-all-substrings-of-a-string-representing-a-number/

Anagram Strings

Two strings will be anagram to each other if and only if they contain the same number of characters and one string can be rearranged to form the other one.

Note - Order of the characters doesn't matter.

- abc and cba are anagram.
- creative and reactive are also anagram

Practice Problem:

https://practice.geeksforgeeks.org/problems/anagram-1587115620/1

Palindrome String

A string is said to be palindrome if the reverse of the string is the same as string. For example, “abcba” is palindrome, but “abcab” is not palindrome.

Practice Problem:

https://practice.geeksforgeeks.org/problems/palindrome-string0817/1

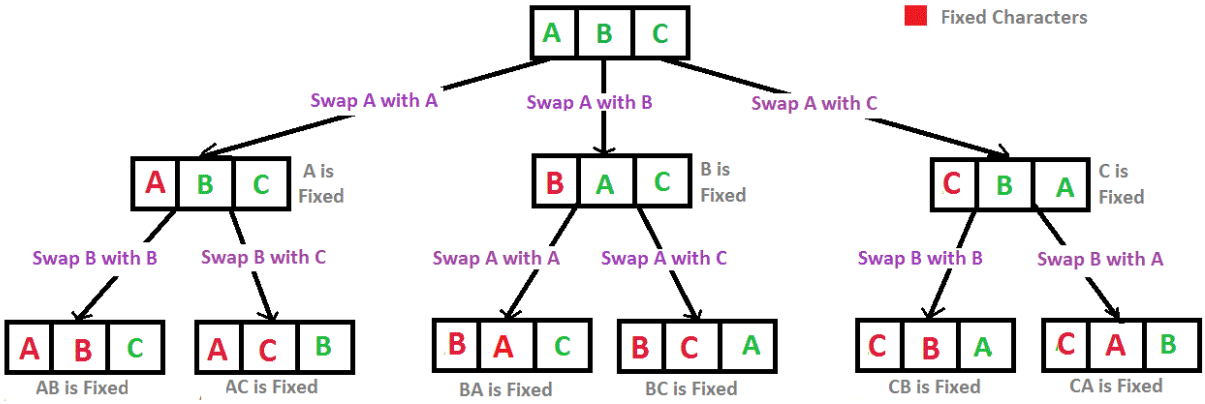
Permutations of a String

To solve this problem, we need to understand the concept of backtracking.

Note: Backtracking will be covered later in another week so if you don’t understand it now, it’s totally fine.

According to the backtracking algorithm:

- Fix a character in the first position and swap the rest of the character with the first character. Like in ABC, in the first iteration three strings are formed: ABC, BAC, and CBA by swapping A with A, B and C respectively.
- Repeat step 1 for the rest of the characters like fixing second character B and so on.
- Now swap again to go back to the previous position. E.g., from ABC, we formed ABC by fixing B again, and we backtrack to the previous position and swap B with C. So, now we got ABC and ACB.
- Repeat these steps for BAC and CBA, to get all the permutations.



Recursion Tree for Permutations of String "ABC"

For code refer :

https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/

Pattern Searching Algorithms

These algorithms are useful in the case of searching a string within another string. Also also referred to as String Searching Algorithms

Naive Approach

Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

```
void search(char* pat, char* txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++) {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i + j] != pat[j])
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            cout << "Pattern found at index "
                 << i << endl;
    }
}
```

The worst case complexity to this naive algorithms is O(m(n-m+1))

Knuth-Morris-Pratt (KMP) Algorithm

The naive algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. For example :

txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n).

The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match. *But how to do that ?*

So to do this we use the Longest Proper Prefix which is Suffix (LPS) array. We process the pattern and prepare an integer array lps[] which tells us the count of the characters to be skipped. lps[i] could be defined as the longest prefix which is also a proper suffix. We need to use it properly at one place to make sure that the whole substring is not considered.

Examples of lps[] construction:
For the pattern "AAAA",
lps[] is [0, 1, 2, 3]

For the pattern "ABCDE",
lps[] is [0, 0, 0, 0, 0]

For the pattern "AABAACAABAA",
lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern "AAACAAAAAC",
lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern "AAABAAA",
lps[] is [0, 1, 2, 0, 1, 2, 3]

We use a value from lps[] to decide the next characters to be matched. The idea is to not match a character that we know will anyway match.

Additional Resources for KMP algorithm:

- https://towardsdatascience.com/pattern-search-with-the-knuth-morris-pratt-kmp-algorithm-8562407dba5b
- https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
- https://www.youtube.com/watch?v=V5-7GzOfADQ&t=925s

Rabin-Karp Algorithm

The Rabin-Karp algorithm slides the pattern one by one, but unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and **if the hash values match then only it starts matching individual characters**. So the algorithm needs to calculate hash values for following strings.

1) Pattern itself.
2) All the substrings of the text of length m.

To efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which can compute the hash at the next shift from the current hash value and next character in text and rehash must be O(1) operation.

Rehashing is done using the following formula:

hash(txt[s+1 .. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q
hash(txt[s .. s+m-1]) : Hash value at shift s.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift s+1)

where, d: Number of characters in the alphabet
q: A prime number
h: d^(m-1)

Example:
Pattern length is 3 and string is “23456”

You compute the value of the first window (which is “234”) as 234.
Now how will you compute the value of the next window “345”?
You will do (234 – 2*100)*10 + 5 and get 345.

For code refer:
https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/

Boyer Moore Algorithm

The Boyer Moore algorithm also preprocesses the pattern. It is a combination of the following two approaches:

1) Bad Character Heuristic
2) Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. It processes the pattern and creates different arrays for each of the two heuristics. At every step, it slides the pattern by the max of the slides suggested by each of the two heuristics. So it uses the greatest offset suggested by the two heuristics at every step. Unlike the previous pattern searching algorithms, this algorithm starts matching from the last character of the pattern.

Check this video to understand the two approaches:
https://www.youtube.com/watch?v=4Xyhb72LCX4

For code refer:
https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/

Additional Resources:

https://www.faceprep.in/c/string-operations-in-c-c-java-and-python/
https://www.geeksforgeeks.org/string-data-structure/
https://www.youtube.com/watch?v=GTJr8OvyEVQ
https://www.youtube.com/watch?v=qQ8vS2btsx