Strings

This C++ file is a comprehensive playbook of fundamental C-style string manipulation algorithms. It serves as a practical, hands-on guide, moving from basic operations to more complex, algorithmic challenges. By exclusively using character arrays and C standard libraries, it provides a low-level perspective on how strings are managed in memory.

The file begins by establishing the basics: creating character arrays, the critical role of the **null terminator (\0)**, calculating string length, and performing case conversions using direct **ASCII value manipulation**. It then progresses to practical text-processing tasks like counting vowels, consonants, and words in a sentence, and validating a string against a set of allowed characters (e.g., for a username).

The core of the file showcases a variety of powerful algorithms. It demonstrates multiple techniques for common problems, providing a clear comparison of different approaches. For instance, it reverses a string using both an efficient in-place swapping method and a simpler auxiliary array method. For finding duplicates, it contrasts a straightforward **hash map** (frequency array) with a highly efficient and memory-saving **bitwise operation** technique.

Finally, it tackles classic computer science problems like determining if two strings are **anagrams** (using a hash map to compare character frequencies) and generating all possible **permutations** of a string using a clever recursive, backtracking algorithm. This file is an excellent resource for anyone looking to build a robust, foundational understanding of string algorithms from the ground up.

**Creating array of charecter**

This section covers the fundamental ways to declare and initialize a C-style string, which is essentially an array of characters. It emphasizes the importance of the null terminator (\0) for properly formatted strings.

* char name2[5] = {'J', 'O', 'H', 'N', '\0'}; This manually initializes a character array. The \0 character, known as the **null terminator**, is crucial. It acts as a sentinel value that tells functions like printf where the string officially ends.
* char arr3[5] = {65, 66, 67, 68, 69}; This demonstrates that characters are stored as **ASCII values** in memory. The number 65 is the ASCII code for the character 'A'.
* gets(name); This is a C function to read a line of text from the user, including spaces, until the user presses Enter. It's a simple way to get string input.

**Finding length of the string**

This code calculates the length of a C-style string by iterating through it until it finds the end.

* for(i; s[i]!='\0'; i++); This is the core of the logic. The loop has no body; its only purpose is to increment the counter i for every character it encounters. The loop **terminates** when it finds the null character \0, at which point i will hold the exact length of the string.

**Converting string to lower case**

This snippet converts an uppercase string to a lowercase one by directly manipulating the ASCII values of its characters.

* s[i]+=32; This line leverages a property of the ASCII table: the ASCII value for any lowercase letter is exactly **32 greater** than its corresponding uppercase letter (e.g., 'A' is 65, 'a' is 97). By adding 32, it effectively shifts the character from uppercase to lowercase.

**Toggle case(some upper, some lower)**

This code "toggles" the case of each character in a string—it converts uppercase letters to lowercase and lowercase letters to uppercase.

* if(s[i] >= 65 && s[i]<=90){ s[i]+=32; } This condition checks if the character's ASCII value falls within the range of **uppercase letters** (65-90). If it does, it adds 32 to convert it to lowercase.
* else if(s[i]>=97 && s[i]<=122){ s[i]-=32; } This condition checks if the character is a **lowercase letter** (97-122). If so, it subtracts 32 to convert it to uppercase.

**Vovel, Consonent counter and counting no of words**

This section performs a basic analysis of a sentence: it counts the number of vowels, consonants, and words.

* if(s[i]=='a' || s[i]=='e' ... || s[i]=='U'){ Vcount++; } This if statement explicitly checks if the current character is one of the ten possible vowel characters (five lowercase, five uppercase).
* else if((s[i] >= 65 && s[i]<=90) || (s[i]>=97 && s[i]<=122)){ Ccount++; } If a character is not a vowel but its ASCII value falls within the range of any letter, it must be a **consonant**.
* if(s[i] == ' ' && s[i-1]!= ' '){ Nwords++; } This is a robust way to count words. It increments the word count only when it encounters a space that is **not preceded by another space**. This correctly handles sentences with multiple spaces between words.

**Validating a string(Checking if string contains any special char)**

This code checks if a string is "valid" by ensuring it contains only alphanumeric characters (letters and numbers) and no special symbols.

* if(!(name[i]>=65 && name[i]<=90) && !(name[i]>=97 && name[i]<=122) && !(name[i]>=48 && name[i]<=57)){ return 0; } This is the core validation logic. The ! operator means "NOT". The condition checks if a character is **NOT** an uppercase letter, **AND NOT** a lowercase letter, **AND NOT** a digit. If a character fails all these checks, it must be a special character, and the function immediately returns 0 (invalid).

**Reversing a string using swap method**

This code snippet reverses a string in-place without using a second array. It's an efficient method that minimizes memory usage.

* for(j = 0; s[j]!='\0'; j++); This first loop simply finds the **length** of the string and positions the pointer j at the null terminator.
* for(i = 0, j--; i<j; i++, j--){ ... } This is the **two-pointer swapping loop**. Pointer i starts at the beginning, and j starts at the end. They move towards the middle (i++, j--). The loop continues as long as i < j.
* k = s[i]; s[i] = s[j]; s[j] = k; Inside the loop, the characters at the i and j positions are **swapped** using a temporary variable k.

**Reversing a string using [auxiliary array] method**

*(The heading was "using swap method," but this code uses an auxiliary array, so the description reflects that.)*

This code reverses a string by using a second, temporary array to build the reversed version.

* char d[j]; An **auxiliary array** d is created with the same size as the original string s.
* for(i = 0, j--; i<length; i++, j--){ d[i] = s[j]; } This loop populates the new array d. It iterates from the beginning of d (using i) and copies characters from the **end** of the source string s (using j).
* d[length] = '\0'; After the loop, the **null terminator** is manually added to the end of the new array d to ensure it is a valid, printable string.

**Finding duplicates in a string using hashmap**

This code efficiently finds which characters are duplicated in a string by using an integer array as a frequency map (or hash map).

* int arr[26] = {0}; This creates our **hash map**. It's an array of 26 integers, one for each letter of the alphabet, all initialized to zero.
* arr[name[i] - 97]++; This is the key step. It maps each character to an array index. Subtracting 97 from a lowercase character's ASCII value ('a' is 97, 'b' is 98, etc.) converts it to a 0-25 index. The value at that index is then incremented, effectively **counting the frequency** of each character.

**Finding duplicates in a string using bitwise operations**

This is a highly efficient and memory-saving technique to find duplicates. It uses a single integer as a bitmask to keep track of which characters have been seen.

* long int H = 0, x = 0; H is our **bitmask**, initialized to zero. It will store our "hash". x is a temporary variable.
* x=x<<(A[i]-97); This line sets a specific bit in x. x starts as 1 (binary ...0001). The left shift << moves this '1' bit to a position corresponding to the character. For 'a', it's shifted 0 times; for 'b', 1 time; for 'c', 2 times, and so on.
* if((x&H)>0){ ... } The bitwise AND & checks if the bit corresponding to the current character is **already set** in our main bitmask H. If it is, (x&H) will be greater than zero, meaning we've seen this character before—it's a duplicate.
* else{ H = x|H; } If it's not a duplicate, the bitwise OR | is used to **set the bit** for the current character in the bitmask H, marking it as "seen".

**Finding weather 2 strings are anagram or not**

This code determines if two strings are anagrams (if they contain the same characters with the same frequencies) using a hash map.

* int H[26] = {0}; A **frequency map** H is created to store the counts of each character.
* for (int i = 0; A[i] != '\0'; i++) { H[A[i] - 'a']++; } The first loop iterates through the first string A and **builds the frequency map**, incrementing the count for each character found.
* for (int i = 0; B[i] != '\0'; i++) { H[B[i] - 'a']--; ... } The second loop iterates through the second string B and **decrements the counts**.
* if (H[B[i] - 'a'] < 0) { ... } If at any point a count drops below zero, it means string B has more of a certain character than string A, so they cannot be anagrams.

**Permutations of a string**

This code uses a recursive, backtracking algorithm to find and print all possible arrangements (permutations) of the characters in a string.

* static int A[10] = {0}; static char res[10]; Two static arrays are used. A is a "flag" array to **track which characters** from the original string have already been used in the current permutation. res is used to **build the permutation string**.
* if(S[k]=='\0'){ printf("%s\n", res); } This is the **base case** for the recursion. If we have reached the end of the string (k is at the null terminator's position), it means we have built a complete permutation, so we print it.
* if(A[i] == 0){ ... } This loop iterates through the original string characters. It only considers a character S[i] if its corresponding flag A[i] is 0 (meaning it's not currently in use).
* res[k] = S[i]; A[i] = 1; perm(S, k+1); A[i] = 0; This is the **recursion and backtracking** step:
  1. res[k] = S[i];: Add the available character to the result.
  2. A[i] = 1;: Mark this character as "used".
  3. perm(S, k+1);: Make a recursive call to find the rest of the permutation.
  4. A[i] = 0;: **This is the backtrack.** After the recursive call returns, un-mark the character, making it available for use in other permutations.