
String

A string, which is composed of a sequence of characters, is quite an important data structure. Many programming languages have special rules for strings for the optimization purpose, some of which are highlighted in C/C++, C#, and Java.

Strings in C/C++

All literal strings in C/C++ end with a special character '\0', so it is easy to find the end of a string. However, there is an extra cost for the special character, and it is easy to make mistakes. This is easy to see in Listing 3-7.

***Listing 3-7.** C/C++ Code for End of a String*

```
   char str[10];
strcpy(str, "0123456789");
```

A character array with length 10 is declared first, and the content of a string "0123456789" is copied into it. It seems that the string "0123456789" only has 10 characters, but its actual length is 11 because there is an extra character '\0' at its end. The length should be at least 11 for a character array to accommodate the string.

Strings in C#

Strings are encapsulated in a class `System.String` in C#, whose contents are immutable. When we try to modify the content of a string, a new instance will be created. There are many interview questions about immutable strings. For example, what is the final content of `str` in the C# code in Listing 3-8?

***Listing 3-8.** C# Code for Immutable Strings*

```
   String str = "hello";
str.ToUpper();
str.Insert(0, " WORLD");
```

Although there are two operations, `ToUpper` and `Insert`, on `str`, it keeps the original content "hello" unchanged. When we try to modify the content of a string, the modified result is in return value.

If there are multiple editing operations on a string, multiple temporary instances will be created, and it has a negative impact on both time and space efficiencies. A new class related to string, `StringBuilder`, is defined to accommodate the modified result. Usually, `StringBuilder` is a better choice if we continue modifying strings many times.

Similar to editing strings, a new instance will be created when we try to assign a literal string to another string. An example is shown in Listing 3-9.

***Listing 3-9.** C# Code to Assign Strings*

```
   void ValueOrReference(Type type) {
    String result = "The type " + type.Name;

    if (type.IsValueType)
        Console.WriteLine(result + " is a value type.");
    else
        Console.WriteLine(result + " is a reference type.");
}

void ModifyString(String text) {
    text = "world";
}

void Main(string[] args) {
    String text = "hello";

       ValueOrReference(text.GetType());
    ModifyString(text);

    Console.WriteLine(text);
}
```

This example checks whether the class `String` is a value type and reference type first. Since it is defined as `public sealed class String {...}`, it is a reference type.

It assigns a new string "world" to `text` in the method `ModifyString`. A new string instance with content "world" is created here, and it is referenced by `text`. The variable `text` references the original string outside the method `ModifyString` because `text` is not marked as `ref` or `out` in the argument list. Therefore, the output for `text` is still "hello". If the expected output is "world", we have to mark the parameter `text` with `ref` or `out`.

Strings in Java

A section of memory is allocated for literal strings in Java. When a string is created once, it can be referenced by other instances. This optimization mechanism avoids recreation, but it makes identity and equality tests more complicated and confusing. Please read the code in Listing 3-10. What is the result?

Listing 3-10. *Java Code for Equality and Identity of Strings*

```
void testEquality() {
 String str1 = "Hello world.";
 String str2 = "Hello world.";

 if (str1 == str2)
     System.out.print("str1 == str2\n");
 else
     System.out.print("str1 != str2\n");

 if(str1.equals(str2))
     System.out.print("str1 equals to str2\n");
 else
     System.out.print("str1 doesn't equal to str2\n");

 String str3 = new String("Hello world.");
 String str4 = new String("Hello world.");

 if (str3 == str4)
     System.out.print("str3 == str4\n");
 else
     System.out.print("str3 != str4\n");

 if(str3.equals(str4))
     System.out.print("str3 equals to str4\n");
 else
     System.out.print("str3 doesn't equal to str4\n");
}
```

When the first line of code `String str1 = "Hello world."` executes, a string "Hello world." is created, and the variable `str1` references to it. Another string "Hello world." will not be created again when the next line of code executes because of optimization. The variable `str2` also references the existing "Hello world.".

The operator `==` checks the identity of the two objects (whether two variables reference the same object). Since `str1` and `str2` reference the same string in memory, they are identical to each other. The method `equals` checks equality of two objects (whether two objects have the same content). Of course, the contents of `str1` and `str2` are the same.

When code `String str3 = new String("Hello world.")` executes, a new instance of `String` with content "Hello world." is created and it is referenced by the variable `str3`. Then another instance of `String` with content "Hello world." is created again, and referenced by `str4`. Since `str3` and `str4` reference two different instances, they are not identical, but their contents are the same.

Therefore, the output contains four lines:

```
str1 == str2
str1 equals to str2
str3 != str4
str3 equals to str4
```

Replace Blanks in a String

Question 9 Please implement a function to replace each blank in a string with "%20". For instance, it outputs "We%20are%20happy." if the input is "We are happy.".

If a URL parameter contains some special characters in web programming, such as blanks and '#'s, it may block servers from retrieving correct parameter information. Therefore, it is necessary to convert these characters into strings understandable by servers. The conversion rule is to append the ASCII value of a special character to a '%'. For example, the ASCII value for a blank is 32, 0x20 in hexadecimal, so a blank is converted to "%20". Take '#' as another example: its ASCII value is 35, 0x23 in hexadecimal, so it is replaced with "%23".

There are library methods to replace a piece of string with another, such as `String.Replace` in C# and `String.replaceAll` in Java. However, usually it is not the interviewers' intention to ask candidates to just call existing methods in libraries. They are more interested to know whether candidates have the ability to implement the replacement functionality.

A string with blanks will become longer if each blank is replaced with three characters: '%', '2', and '0'. If the replacement is on the original string, the longer converted string may overlap with memory behind the string. If it is allowed to create a new string and replace blanks on the new string, we can allocate enough memory to accommodate the longer converted string. Since there are two options, candidates should ask interviewers to clarify their requirements. Let's suppose the requirement is to replace blanks on the original string, and there is enough vacant memory behind it.

Replace from Left to Right in O(n^2) Time

An intuitive solution is to scan a string from beginning to end and replace each blank when it is met. Because a character is replaced with three, we must move characters behind blanks; otherwise, two characters will be overlapped.

Let's take a string " We are happy." as an example. We employ grids to visualize characters in a string, as shown in Figure 3-3(a).

When the first blank is replaced, the string becomes the content of Figure 3-3(b). We have to move the characters in the area in the light gray background. When the second blank is replaced, it becomes the content in Figure 3-3(c). Note that characters in " happy." are moved twice.

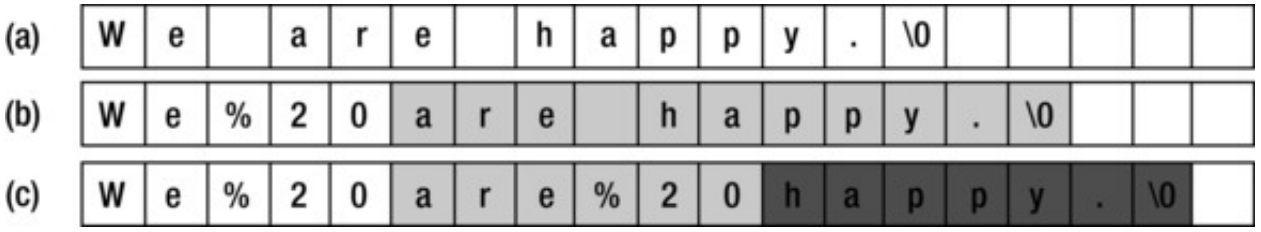

Figure 3-3. *Replace every blank in "We are happy." with "%20" from left to right. (a) This is the original string, "We are happy.". (b) Replace the first blank with "%20". It requires you to move characters with the light gray background. (c) Replace the second blank with "%20". It requires you to move characters with the dark gray background again.*

Supposing the length of the string is n. O(n) characters are moved for each blank, so the total time efficiency is O(n^2) to replace O(n) blanks.

Interviewers may not be satisfied when they are told of this solution. What they expect is a more efficient solution. In the previous analysis, there are many characters moved multiple times. Is it possible to reduce the number of movements? Fortunately, the answer is yes. Let's have a try by replacing blanks from the end to the beginning.

Replace from Right to Left in O(n) Time

The number of blanks in the original string is gotten when it is scanned, and then the length of the converted string can also be gotten. The length increases by two when a blank is replaced, so the length after conversion is obtained by adding the original length to double the number of blanks. Take the string " We are happy." as an example again. Its length is 14 (including '\0'), and the length of the replaced string is 18 since there are two blanks in the original string.

Characters are copied or replaced from right to left at this time. Two pointers, P_1 and P_2, are declared. P_1 is initialized to the end of original string, and P_2 is initialized to the end of replaced string, as shown in Figure 3-4(a). The character pointed to by P_1 is copied to where P_2 points, and both pointers are moved backward until P_1 points to a blank (Figure 3-4(b)). Characters with the light gray background are copied. When a blank is met, P_1 moves backward one cell. Three characters "_" are inserted to where P_2 points, and then P_2 moves backward three cells, as shown in Figure 3-4(c).

This process continues until it meets a blank again (Figure 3-4(d)). Similarly, P_1 moves backward one cell. Three characters "_" are inserted to where P_2 points, and then P_2 moves backward three cells (Figure 3-4(e)). All blanks have been replaced because P_1 and P_2 overlap with each other.

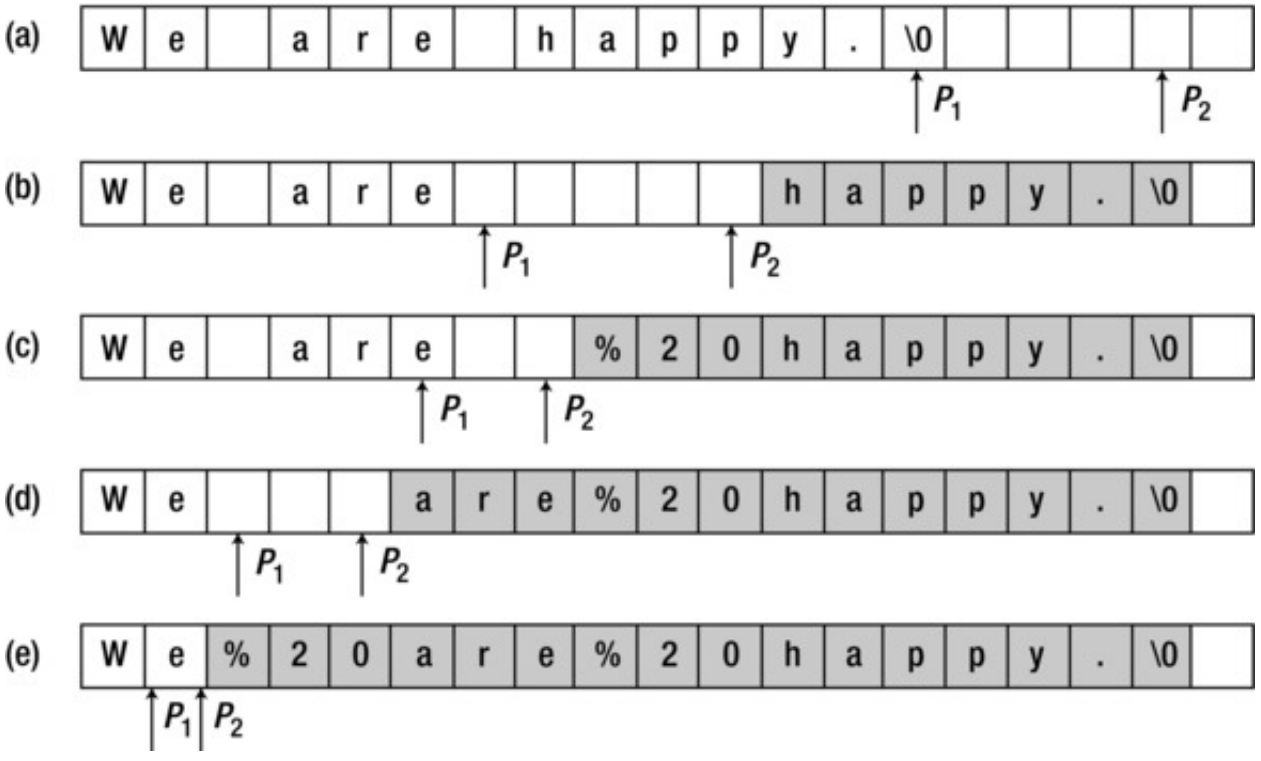

Figure 3-4. *Replace every blank in "We are happy." with "_" backward. (a)* P_1 *is initialized to the end of the original string, and* P_2 *is initialized to the end of the replaced string. (b) Copy the character pointed to by* P_1 *to where* P_2 *points until the first blank is met. (c) "_" is inserted where* P_2 *points.* P_1 *moves one cell, and* P_2 *moves three cells backward. (d) Copy the character pointed to by* P_1 *to where* P_2 *points until the second blank is met. (e) "_" is inserted to where* P_2 *points.* P_1 *moves one cell, and* P_2 *moves three cells backward. All blanks have been replaced because* P_1 *overlaps with* P_2.

Note that all characters are copied (moved) once at most, so its time efficiency is O(n) and it is faster than the first solution.

It is time to implement code when your solution is accepted by interviewers. Some sample code is shown in Listing 3-11.

Listing 3-11. *C Code to Replace Blanks with "%20"*

```
   /*capacity is total capacity of a string, which is longer than its actual length*/
void ReplaceBlank(char string[], int capacity) {
    int originalLength, numberOfBlank, newLength;
    int i, indexOfOriginal, indexOfNew;

    if(string == NULL || capacity <= 0)
        return;

    /*originalLength is the actual length of string*/
    originalLength = numberOfBlank = i = 0;
    while(string[i] != '\0') {
        ++ originalLength;

        if(string[i] == ' ')
            ++ numberOfBlank;

        ++ i;
    }

       /*newLength is the length of the replaced string*/
    newLength = originalLength + numberOfBlank * 2;
    if(newLength > capacity)
        return;

    indexOfOriginal = originalLength;
    indexOfNew = newLength;
    while(indexOfOriginal >= 0 && indexOfNew > indexOfOriginal) {
        if(string[indexOfOriginal] == ' ') {
            string[indexOfNew --] = '0';
            string[indexOfNew --] = '2';
            string[indexOfNew --] = '%';
        }
        else {
            string[indexOfNew --] = string[indexOfOriginal];
        }

        -- indexOfOriginal;
    }
}
```

Source Code:

009_ReplaceBlanks.c

Test Cases:

- A string contains some blanks (including cases where some blanks are at the beginning or end of a string, or inside a string; some blanks are continuous)
- A string does not contain any blanks
- Special strings (such as empty strings, a string with only a blank, a string with only some continuous blanks, the input string pointer is `NULL`)

Question 10 Given two sorted arrays, denoted as *array1* and *array2*, please merge them into *array1* and keep the merged array sorted. Suppose there is sufficient vacant memory at the end of *array1* to accommodate elements of *array2*.

All elements in an array are sequential, so some elements will be shifted when a new element is inserted. Supposing *array1* has *m* elements and *array2* has *n* elements. Since O(*m*) elements will be shifted when an element of *array2* is inserted to *array1*, it costs O(*mn*) time to merge two arrays via insertions.

Inspired by the solution of the previous problem, it is better to copy and move elements from right to left. The last two elements of these two arrays are compared, and the greater one is copied to the location with index (*m*+*n*-1). It continues to compare and copy until no numbers in *array2* are left. Sample code in C is shown in Listing 3-12.

Listing 3-12. C Code to Merge Sorted Arrays

```
   // Supposing there is enough memory at the end of array1,
// in order to accommodate numbers in array2
void merge(int* array1, int length1, int* array2, int length2) {
    int index1, index2, indexMerged;

    if(array1 == NULL || array2 == NULL)
        return;

    index1 = length1 - 1;
    index2 = length2 - 1;
    indexMerged = length1 + length2 - 1;

    while(index1 >= 0 && index2 >= 0) {
        if(array1[index1] >= array2[index2])
            array1[indexMerged--] = array1[index1--];
        else
            array1[indexMerged--] = array2[index2--];
    }

    while(index2 >= 0)
        array1[indexMerged--] = array2[index2--];
}
```

Since only one element in *array1* or *array2* is copied and moved once in each step, the overall time complexity is O(*m*+*n*).

Source Code:

```
010_MergeSortedArrays.c
```

Test Cases:

- Merge two sorted arrays (including cases where there are duplicated numbers in two arrays)
- Special arrays (including cases where one or two pointers to arrays are `NULL`)

Tip If each element is shifted multiple times while merging two arrays (strings) from left to right, it may improve performance if elements are copied and moved from right to left.

String Matching

Regular expressions are an important topic in text processing, and many programming languages provide libraries to support them. For example, Java has a package `java.util.regex` and C# has a namespace `System.Text.RegularExpressions` for regular expressions. However, interviewers usually disallow candidates to employ library utilities to solve problems related to regular expressions, and candidates have to implement matching mechanisms to demonstrate their coding capabilities.

Question 11 How do you implement a function to match regular expressions with '.' and '*' in patterns? The character '.' in a pattern matches a single character, and '*' matches zero or any number of characters preceding it. Matching means that a string fully matches the pattern where all characters in a string match the whole pattern. For example, the string "aaa" matches the pattern "a.a" and the pattern "ab*ac*a". However, it does not match the pattern "aa.a" nor "ab*a".

Our solution matches a character after another from a string and a pattern. Let's first analyze how to match a character. When the character *ch* in the pattern is a '.', it matches whatever character is in the string. If the character *ch* is not a '.' and the character in the string is *ch*, they match each other. When the first characters in a string and a pattern are matched, we continue to match the remaining string and pattern.

It is easy to match when the second character in the remaining pattern is not a '*'. If the first character in the remaining string matches the first character in the remaining pattern, it advances the string and pattern and continues to match next characters; otherwise, it returns `false` directly.

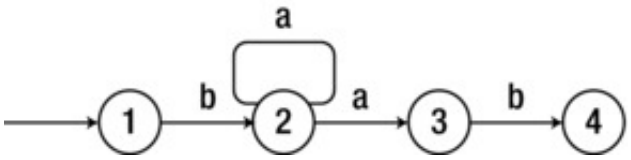

Figure 3-5. *The nondeterministic finite automaton for the pattern ba*ab. There are two choices when it enters the state 2 with an input 'a': it advances to the state 3 or returns back to the state 2.*

It is more complex when the second character in the remaining pattern is a '*' because there might be multiple matching choices. One choice is to advance the pattern by two characters because a '*' may match zero characters in a string. In such a case, a '*' and its preceding character are ignored. If the first character in the remaining string matches the character before the '*' in the pattern, it may advance forward in the string and it has two choices in the pattern: it may advance the pattern by two characters or keep the pattern unchanged.

As shown in the nondeterministic finite automaton of Figure 3-5, it has two choices in the state 2 with an input 'a': it may advance to the state 3 (advance on the pattern by two characters) or return back to state 2 (keeping the pattern unchanged for the next round of matching).

The solution can be implemented based on recursion, as shown in Listing 3-13.

Listing 3-13. *C++ Code for Simple Regular Expression Matching*

```
bool match(char* string, char* pattern) {
    if(string == NULL || pattern == NULL)
        return false;

    return matchCore(string, pattern);
}

bool matchCore(char* string, char* pattern) {
    if(*string == '\0' && *pattern == '\0')
        return true;

    if(*string != '\0' && *pattern == '\0')
        return false;

    if(*(pattern + 1) == '*') {
        if(*pattern == *string || (*pattern == '.' && *string != '\0'))
                // move on the next state
            return matchCore(string + 1, pattern + 2)
                // stay on the current state
                || matchCore(string + 1, pattern)
                // ignore a '*'
                || matchCore(string, pattern + 2);
        else
                // ignore a '*'
            return matchCore(string, pattern + 2);
    }

    if(*string == *pattern || (*pattern == '.' && *string != '\0'))
        return matchCore(string + 1, pattern + 1);

    return false;
}
```

Source Code:

`011_SimpleRegularExpression.cpp`

Test Cases:

- A string matches or does not match a pattern (including cases where there are '.' and/or '*' in the pattern)
- Special inputs (including cases where a string or pattern is empty, pointers to a sting or pattern are `NULL`, or an invalid pattern with '*' is at the beginning)

Question 12 How do you check whether a string stands for a number or not? Numbers include positive and negative integers and floats. For example, strings " +100.", " 5e2", " -.123", " 3.1416", and " -1E-16" stand for numbers, but " 12e", " 1a3.14", " 1.2.3", " +-5", and " 12e+5.4" do not.

A numeric string follows this format:

```
[sign]integral-digits[.[fractional-digits]][e|E[sign]exponential-digits].
```

Elements in square brackets '[' and ']' are optional. The *sign* element is a negative sign symbol ('-') or a positive sign symbol ('+'). There is only one leading sign at most. The *integral-digits* element is a series of digits ranging from 0 to 9 that specify the integral part of the number. It can be absent if the string contains the *fractional-digits* element, which is also a series of digits ranging from 0 to 9 that specify the fractional part of a number. The 'e' or 'E' character indicates that the value is represented in exponential (scientific) notation, which is another series of digits ranging from 0 to 9 specified in *exponential-digits*.

First, it checks whether the leading character is the positive or negative sign symbol, and then it moves on to check the following substring. If the number in the string is a float value, there is a floating point ('.'). Additionally, there might be an exponential notation at the end of an integer or a float number if there is an 'e' or 'E' in the string. The overall process to verify a numeric string is shown in Listing 3-14.

Listing 3-14. *C++ Code to Verify Numeric Strings*

```
bool isNumeric(char* string) {
 if(string == NULL)
     return false;

 if(*string == '+' || *string == '-')
     ++string;
 if(*string == '\0')
     return false;

 bool numeric = true;

 scanDigits(&string);
 if(*string != '\0') {
     // for floats
     if(*string == '.') {
         ++string;
         scanDigital(&string);

         if(*string == 'e' || *string == 'E')
             numeric = isExponential(&string);
     }
     // for integers
     else if(*string == 'e' || *string == 'E')
         numeric = isExponential(&string);
     else
         numeric = false;
 }

 return numeric && *string == '\0';
}
```

The function `scanDigits` scans a segment of a string that only contains digital characters ranging from '0' to '9', as implemented in Listing 3-15.

Listing 3-15. *C++ Code to Scan Digits*

```
void scanDigits(char** string) {
 while(**string != '\0' && **string >= '0' && **string <= '9')
     ++(*string);
}
```

The function `isExponential` is to verify exponential notation at the end of a string. Exponential notation begins with an 'e' or 'E' and may have a sign symbol following it. Therefore, the function `isExponential` can be implemented, as shown in Listing 3-16.

Listing 3-16. *C++ Code to Verify an Exponential Notation*

```
bool isExponential(char** string) {
 if(**string != 'e' && **string != 'E')
```

```
            return false;


        ++(*string);
        if(**string == '+' || **string == '-')
            ++(*string);


        if(**string == '\0')
            return false;


        scanDigits(string);
        return (**string == '\0') ? true : false;
    }
```

Source Code:

012_NumericStrings.cpp

Test Cases:

- Numeric/Non-numeric strings with/without a sign symbol
- Numeric/Non-numeric strings with/without some fractional digits
- Numeric/Non-numeric strings with/without exponential notation
- Special inputs (including cases where a string is empty, the input pointer to a sting is `NULL`)