# 5、字符串

# (07)－翻转句子中单词的顺序

题目：输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。

例如输入“I am a student.”，则输出“student. a am I”。

分析：由于编写字符串相关代码能够反映程序员的编程能力和编程习惯，与字符串相关的问题一直是程序员笔试、面试题的热门题目。本题也曾多次受到包括微软在内的大量公司的青睐。

由于本题需要翻转句子，我们先颠倒句子中的所有字符。这时，不但翻转了句子中单词的顺序，而且单词内字符也被翻转了。我们再颠倒每个单词内的字符。由于单词内的字符被翻转两次，因此顺序仍然和输入时的顺序保持一致。

还是以上面的输入为例子。翻转“I am a student.”中所有字符得到“.tneduts a ma I”，再翻转每个单词中字符的顺序得到“students. a am I”，正是符合要求的输出。

参考代码：

///////////////////////////////////////////////////////////////////////

// Reverse a string between two pointers

// Input: pBegin - the begin pointer in a string

// pEnd - the end pointer in a string

///////////////////////////////////////////////////////////////////////

void Reverse(char \*pBegin, char \*pEnd)

{

if(pBegin == NULL || pEnd == NULL)

return;

while(pBegin < pEnd)

{

char temp = \*pBegin;

\*pBegin = \*pEnd;

\*pEnd = temp;

pBegin ++, pEnd --;

}

}

///////////////////////////////////////////////////////////////////////

// Reverse the word order in a sentence, but maintain the character

// order inside a word

// Input: pData - the sentence to be reversed

///////////////////////////////////////////////////////////////////////

char\* ReverseSentence(char \*pData)

{

if(pData == NULL)

return NULL;

char \*pBegin = pData;

char \*pEnd = pData;

while(\*pEnd != '\0')

pEnd ++;

pEnd--;

// Reverse the whole sentence

Reverse(pBegin, pEnd);

// Reverse every word in the sentence

pBegin = pEnd = pData;

while(\*pBegin != '\0')

{

if(\*pBegin == ' ')

{

pBegin ++;

pEnd ++;

continue;

}

// A word is between with pBegin and pEnd, reverse it

else if(\*pEnd == ' ' || \*pEnd == '\0')

{

Reverse(pBegin, --pEnd);

pBegin = ++pEnd;

}

else

{

pEnd ++;

}

}

return pData;

}

# (17)－把字符串转换成整数

题目：输入一个表示整数的字符串，把该字符串转换成整数并输出。例如输入字符串"345"，则输出整数345。 分析：这道题尽管不是很难，学过C/C++语言一般都能实现基本功能，但不同程序员就这道题写出的代码有很大区别，可以说这道题能够很好地反应出程序员的思维和编程习惯，因此已经被包括微软在内的多家公司用作面试题。建议读者在往下看之前自己先编写代码，再比较自己写的代码和下面的参考代码有哪些不同。

首先我们分析如何完成基本功能，即如何把表示整数的字符串正确地转换成整数。还是以"345"作为例子。当我们扫描到字符串的第一个字符'3'时，我们不知道后面还有多少位，仅仅知道这是第一位，因此此时得到的数字是3。当扫描到第二个数字'4'时，此时我们已经知道前面已经一个3了，再在后面加上一个数字4，那前面的3相当于30，因此得到的数字是3\*10+4=34。接着我们又扫描到字符'5'，我们已经知道了'5'的前面已经有了34，由于后面要加上一个5，前面的34就相当于340了，因此得到的数字就是34\*10+5=345。

分析到这里，我们不能得出一个转换的思路：每扫描到一个字符，我们把在之前得到的数字乘以10再加上当前字符表示的数字。这个思路用循环不难实现。

由于整数可能不仅仅之含有数字，还有可能以'+'或者'-'开头，表示整数的正负。因此我们需要把这个字符串的第一个字符做特殊处理。如果第一个字符是'+'号，则不需要做任何操作；如果第一个字符是'-'号，则表明这个整数是个负数，在最后的时候我们要把得到的数值变成负数。

接着我们试着处理非法输入。由于输入的是指针，在使用指针之前，我们要做的第一件是判断这个指针是不是为空。如果试着去访问空指针，将不可避免地导致程序崩溃。另外，输入的字符串中可能含有不是数字的字符。每当碰到这些非法的字符，我们就没有必要再继续转换。最后一个需要考虑的问题是溢出问题。由于输入的数字是以字符串的形式输入，因此有可能输入一个很大的数字转换之后会超过能够表示的最大的整数而溢出。

现在已经分析的差不多了，开始考虑编写代码。首先我们考虑如何声明这个函数。由于是把字符串转换成整数，很自然我们想到：

int StrToInt(const char\* str);

这样声明看起来没有问题。但当输入的字符串是一个空指针或者含有非法的字符时，应该返回什么值呢？0怎么样？那怎么区分非法输入和字符串本身就是”0”这两种情况呢？

接下来我们考虑另外一种思路。我们可以返回一个布尔值来指示输入是否有效，而把转换后的整数放到参数列表中以引用或者指针的形式传入。于是我们就可以声明如下：

bool StrToInt(const char \*str, int& num);

这种思路解决了前面的问题。但是这个函数的用户使用这个函数的时候会觉得不是很方便，因为他不能直接把得到的整数赋值给其他整形变脸，显得不够直观。

前面的第一种声明就很直观。如何在保证直观的前提下当碰到非法输入的时候通知用户呢？一种解决方案就是定义一个全局变量，每当碰到非法输入的时候，就标记该全局变量。用户在调用这个函数之后，就可以检验该全局变量来判断转换是不是成功。

下面我们写出完整的实现代码。参考代码：

enum Status {kValid = 0, kInvalid};

int g\_nStatus = kValid;

///////////////////////////////////////////////////////////////////////

// Convert a string into an integer

///////////////////////////////////////////////////////////////////////

int StrToInt(const char\* str)

{

g\_nStatus = kInvalid;

long long num = 0;

if(str != NULL)

{

const char\* digit = str;

// the first char in the string maybe '+' or '-'

bool minus = false;

if(\*digit == '+')

digit ++;

else if(\*digit == '-')

{

digit ++;

minus = true;

}

// the remaining chars in the string

while(\*digit != '\0')

{

if(\*digit >= '0' && \*digit <= '9')

{

num = num \* 10 + (\*digit - '0');

// overflow

if(num > std::numeric\_limits<int>::max())

{

num = 0;

break;

}

digit ++;

}

// if the char is not a digit, invalid input

else

{

num = 0;

break;

}

}

if(\*digit == '\0')

{

g\_nStatus = kValid;

if(minus)

num = 0 - num;

}

}

return static\_cast<int>(num);

}

讨论：在参考代码中，我选用的是第一种声明方式。不过在面试时，我们可以选用任意一种声明方式进行实现。但当面试官问我们选择的理由时，我们要对两者的优缺点进行评价。第一种声明方式对用户而言非常直观，但使用了全局变量，不够优雅；而第二种思路是用返回值来表明输入是否合法，在很多API中都用这种方法，但该方法声明的函数使用起来不够直观。

最后值得一提的是，在C语言提供的库函数中，函数atoi能够把字符串转换整数。它的声明是int atoi(const char \*str)。该函数就是用一个全局变量来标志输入是否合法的。

# (21)－左旋转字符串

题目：定义字符串的左旋转操作：把字符串前面的若干个字符移动到字符串的尾部。如把字符串abcdef左旋转2位得到字符串cdefab。请实现字符串左旋转的函数。要求时间对长度为n的字符串操作的复杂度为O(n)，辅助内存为O(1)。

分析：如果不考虑时间和空间复杂度的限制，最简单的方法莫过于把这道题看成是把字符串分成前后两部分，通过旋转操作把这两个部分交换位置。于是我们可以新开辟一块长度为n+1的辅助空间，把原字符串后半部分拷贝到新空间的前半部分，在把原字符串的前半部分拷贝到新空间的后半部分。不难看出，这种思路的时间复杂度是O(n)，需要的辅助空间也是O(n)。

接下来的一种思路可能要稍微麻烦一点。我们假设把字符串左旋转m位。于是我们先把第0个字符保存起来，把第m个字符放到第0个的位置，在把第2m个字符放到第m个的位置…依次类推，一直移动到最后一个可以移动字符，最后在把原来的第0个字符放到刚才移动的位置上。接着把第1个字符保存起来，把第m+1个元素移动到第1个位置…重复前面处理第0个字符的步骤，直到处理完前面的m个字符。

该思路还是比较容易理解，但当字符串的长度n不是m的整数倍的时候，写程序会有些麻烦，感兴趣的朋友可以自己试一下。由于下面还要介绍更好的方法，这种思路的代码我就不提供了。

我们还是把字符串看成有两段组成的，记位XY。左旋转相当于要把字符串XY变成YX。我们先在字符串上定义一种翻转的操作，就是翻转字符串中字符的先后顺序。把X翻转后记为XT。显然有(XT)T=X。

我们首先对X和Y两段分别进行翻转操作，这样就能得到XTYT。接着再对XTYT进行翻转操作，得到(XTYT)T=(YT)T(XT)T=YX。正好是我们期待的结果。

分析到这里我们再回到原来的题目。我们要做的仅仅是把字符串分成两段，第一段为前面m个字符，其余的字符分到第二段。再定义一个翻转字符串的函数，按照前面的步骤翻转三次就行了。时间复杂度和空间复杂度都合乎要求。

参考代码如下：

#include "string.h"

///////////////////////////////////////////////////////////////////////

// Move the first n chars in a string to its end

///////////////////////////////////////////////////////////////////////

char\* LeftRotateString(char\* pStr, unsigned int n)

{

if(pStr != NULL)

{

int nLength = static\_cast<int>(strlen(pStr));

if(nLength > 0 || n == 0 || n > nLength)

{

char\* pFirstStart = pStr;

char\* pFirstEnd = pStr + n - 1;

char\* pSecondStart = pStr + n;

char\* pSecondEnd = pStr + nLength - 1;

// reverse the first part of the string

ReverseString(pFirstStart, pFirstEnd);

// reverse the second part of the strint

ReverseString(pSecondStart, pSecondEnd);

// reverse the whole string

ReverseString(pFirstStart, pSecondEnd);

}

}

return pStr;

}

///////////////////////////////////////////////////////////////////////

// Reverse the string between pStart and pEnd

///////////////////////////////////////////////////////////////////////

void ReverseString(char\* pStart, char\* pEnd)

{

if(pStart == NULL || pEnd == NULL)

{

while(pStart <= pEnd)

{

char temp = \*pStart;

\*pStart = \*pEnd;

\*pEnd = temp;

pStart ++;

pEnd --;

}

}

}

# (36) 在字符串中删除特定的字符

题目：输入两个字符串，从第一字符串中删除第二个字符串中所有的字符。例如，输入”They are students.” 和”aeiou” ，则删除之后的第一个字符串变成”Thy r stdnts.” 。

分析：这是一道微软面试题。在微软的常见面试题中，与字符串相关的题目占了很大的一部分，因为写程序操作字符串能很好的反映我们的编程基本功。

要编程完成这道题要求的功能可能并不难。毕竟，这道题的基本思路就是在第一个字符串中拿到一个字符，在第二个字符串中查找 一下，看它是不是在第二个字符串中。如果在的话，就从第一个字符串中删除。但如何能够把效率优化到让人满意的程度，却也不是一件容易的事情。也就是说，如 何在第一个字符串中删除一个字符，以及如何在第二字符串中查找一个字符，都是需要一些小技巧的。

首先我们考虑如何在字符串中删除一个字符。由于字符串的内存分配方式是连续分配的。我们从字符串当中删除一个字符，需要把后面所有的字符往前移动一个字节的位置。但如果每次删除都需要移动字符串后面的字符的话，对于一个长度为n 的字符串而言，删除一个字符的时间复杂度为O(n) 。而对于本题而言，有可能要删除的字符的个数是n ，因此该方法就删除而言的时间复杂度为O(n2) 。

事实上，我们并不需要在每次删除一个字符的时候都去移动后面所有的字符。我们可以设想，当一个字符需要被删除的时候，我们把它所占的位置让它后面的字符来填补，也就相当于这个字符被删除了。在具体实现中，我们可以定义两个指针(pFast 和pSlow) ，初始的时候都指向第一字符的起始位置。当pFast 指向的字符是需要删除的字符，则pFast 直接跳过，指向下一个字符。如果pFast 指向的字符是不需要删除的字符，那么把pFast 指向的字符赋值给pSlow 指向的字符，并且pFast 和pStart 同时向后移动指向下一个字符。这样，前面被pFast 跳过的字符相当于被删除了。用这种方法，整个删除在O(n) 时间内就可以完成。

接下来我们考虑如何在一个字符串中查找一个字符。当然，最简单的办法就是从头到尾扫描整个字符串。显然，这种方法需要一个循环，对于一个长度为n 的字符串，时间复杂度是O(n) 。

由于字符的总数是有限的。对于八位的char 型字符而言，总共只有28=256 个字符。我们可以新建一个大小为256 的数组，把所有元素都初始化为0 。然后对于字符串中每一个字符，把它的ASCII 码映射成索引，把数组中该索引对应的元素设为１。这个时候，要查找一个字符就变得很快了：根据这个字符的ASCII 码，在数组中对应的下标找到该元素，如果为0 ，表示字符串中没有该字符，否则字符串中包含该字符。此时，查找一个字符的时间复杂度是O(1) 。其实，这个数组就是一个hash 表。这种思路的详细说明，详见 本面试题系列的第13 题 。

基于上述分析，我们可以写出如下代码：

///////////////////////////////////////////////////////////////////////

// Delete all characters in pStrDelete from pStrSource

///////////////////////////////////////////////////////////////////////

void DeleteChars(char\* pStrSource, const char\* pStrDelete)

{

if(NULL == pStrSource || NULL == pStrDelete)

return;

// Initialize an array, the index in this array is ASCII value.

// All entries in the array, whose index is ASCII value of a

// character in the pStrDelete, will be set as 1.

// Otherwise, they will be set as 0.

const unsigned int nTableSize = 256;

int hashTable[nTableSize];

memset(hashTable, 0, sizeof(hashTable));

const char\* pTemp = pStrDelete;

while ('\0' != \*pTemp)

{

hashTable[\*pTemp] = 1;

++ pTemp;

}

char\* pSlow = pStrSource;

char\* pFast = pStrSource;

while ('\0' != \*pFast)

{

// if the character is in pStrDelete, move both pStart and

// pEnd forward, and copy pEnd to pStart.

// Otherwise, move only pEnd forward, and the character

// pointed by pEnd is deleted

if(1 != hashTable[\*pFast])

{

\*pSlow = \*pFast;

++ pSlow;

}

++pFast;

}

\*pSlow = '\0';

}

# (46)－对称子字符串的最大长度[算法]

题目：输入一个字符串，输出该字符串中对称的子字符串的最大长度。比如输入字符串“google”，由于该字符串里最长的对称子字符串是“goog”，因此输出4。

分析：可能很多人都写过判断一个字符串是不是对称的函数，这个题目可以看成是该函数的加强版。

**引子：判断字符串是否对称**

要判断一个字符串是不是对称的，不是一件很难的事情。我们可以先得到字符串首尾两个字符，判断是不是相等。如果不相等，那该字符串肯定不是对称的。否则我们接着判断里面的两个字符是不是相等，以此类推。基于这个思路，我们不难写出如下代码：

////////////////////////////////////////////////////////////////

// Whether a string between pBegin and pEnd is symmetrical?

////////////////////////////////////////////////////////////////

bool IsSymmetrical(char\* pBegin, char\* pEnd)

{

       if(pBegin == NULL || pEnd == NULL || pBegin > pEnd)

              return false;

       while(pBegin < pEnd)

       {

              if(\*pBegin != \*pEnd)

                     return false;

              pBegin++;

              pEnd --;

       }

       return true;

}

要判断一个字符串pString是不是对称的，我们只需要调用IsSymmetrical(pString, &pString[strlen(pString) – 1])就可以了。

**解法一：O（n3）的算法**

现在我们试着来得到对称子字符串的最大长度。最直观的做法就是得到输入字符串的所有子字符串，并逐个判断是不是对称的。如果一个子字符串是对称的，我们就得到它的长度。这样经过比较，就能得到最长的对称子字符串的长度了。于是，我们可以写出如下代码：

////////////////////////////////////////////////////////////////

// Get the longest length of its all symmetrical substrings

// Time needed is O(T^3)

////////////////////////////////////////////////////////////////

int GetLongestSymmetricalLength\_1(char\* pString)

{

       if(pString == NULL)

              return 0;

       int symmeticalLength = 1;

       char\* pFirst = pString;

       int length = strlen(pString);

       while(pFirst < &pString[length - 1])

       {

              char\* pLast = pFirst + 1;

              while(pLast <= &pString[length - 1])

              {

                     if(IsSymmetrical(pFirst, pLast))

                     {

                           int newLength = pLast - pFirst + 1;

                           if(newLength > symmeticalLength)

                                  symmeticalLength = newLength;

                     }

                     pLast++;

              }

              pFirst++;

       }

       return symmeticalLength;

}

我们来分析一下上述方法的时间效率。由于我们需要两重while循环，每重循环需要O（n）的时间。另外，我们在循环中调用了IsSymmetrical，每次调用也需要O（n）的时间。因此整个函数的时间效率是O（n3）。

通常O（n3）不会是一个高效的算法。如果我们仔细分析上述方法的比较过程，我们就能发现其中有很多重复的比较。假设我们需要判断一个子字符串具有aAa的形式（A是aAa的子字符串，可能含有多个字符）。我们先把pFirst指向最前面的字符a，把pLast指向最后面的字符a，由于两个字符相同，我们在IsSymtical函数内部向后移动pFirst，向前移动pLast，以判断A是不是对称的。接下来若干步骤之后，由于A也是输入字符串的一个子字符串，我们需要再一次判断它是不是对称的。也就是说，我们重复多次地在判断A是不是对称的。

造成上述重复比较的根源在于IsSymmetrical的比较是从外向里进行的。在判断aAa是不是对称的时候，我们不知道A是不是对称的，因此需要花费O（n）的时间来判断。下次我们判断A是不是对称的时候，我们仍然需要O（n）的时间。

**解法二：O（n2）的算法**

如果我们换一种思路，我们从里向外来判断。也就是我们先判断子字符串A是不是对称的。如果A不是对称的，那么向该子字符串两端各延长一个字符得到的字符串肯定不是对称的。如果A对称，那么我们只需要判断A两端延长的一个字符是不是相等的，如果相等，则延长后的字符串是对称的。因此在知道A是否对称之后，只需要O（1）的时间就能知道aAa是不是对称的。

我们可以根据从里向外比较的思路写出如下代码：

////////////////////////////////////////////////////////////////

// Get the longest length of its all symmetrical substrings

// Time needed is O(T^2)

////////////////////////////////////////////////////////////////

int GetLongestSymmetricalLength\_2(char\* pString)

{

       if(pString == NULL)

              return 0;

       int symmeticalLength = 1;

       char\* pChar = pString;

       while(\*pChar != '\0')

       {

              // Substrings with odd length

              char\* pFirst = pChar - 1;

              char\* pLast = pChar + 1;

              while(pFirst >= pString && \*pLast != '\0' && \*pFirst == \*pLast)

              {

                     pFirst--;

                     pLast++;

              }

              int newLength = pLast - pFirst - 1;

              if(newLength > symmeticalLength)

                     symmeticalLength = newLength;

              // Substrings with even length

              pFirst = pChar;

              pLast = pChar + 1;

              while(pFirst >= pString && \*pLast != '\0' && \*pFirst == \*pLast)

              {

                     pFirst--;

                     pLast++;

              }

              newLength = pLast - pFirst - 1;

              if(newLength > symmeticalLength)

                     symmeticalLength = newLength;

              pChar++;

       }

       return symmeticalLength;

}

由于子字符串的长度可能是奇数也可能是偶数。长度是奇数的字符串是从只有一个字符的中心向两端延长出来，而长度为偶数的字符串是从一个有两个字符的中心向两端延长出来。因此我们的代码要把这种情况都考虑进去。在上述代码中，我们从字符串的每个字符串两端开始延长，如果当前的子字符串是对称的，再判断延长之后的字符串是不是对称的。由于总共有O（n）个字符，每个字符可能延长O（n）次，每次延长时只需要O（1）就能判断出是不是对称的，因此整个函数的时间效率是O（n2）。

# (28)-字符串的排列

题目：输入一个字符串，打印出该字符串中字符的所有排列。例如输入字符串abc，则输出由字符a、b、c所能排列出来的所有字符串abc、acb、bac、bca、cab和cba。

分析：这是一道很好的考查对递归理解的编程题，因此在过去一年中频繁出现在各大公司的面试、笔试题中。

我们以三个字符abc为例来分析一下求字符串排列的过程。首先我们固定第一个字符a，求后面两个字符bc的排列。当两个字符bc的排列求好之后，我们把第一个字符a和后面的b交换，得到bac，接着我们固定第一个字符b，求后面两个字符ac的排列。现在是把c放到第一位置的时候了。记住前面我们已经把原先的第一个字符a和后面的b做了交换，为了保证这次c仍然是和原先处在第一位置的a交换，我们在拿c和第一个字符交换之前，先要把b和a交换回来。在交换b和a之后，再拿c和处在第一位置的a进行交换，得到cba。我们再次固定第一个字符c，求后面两个字符b、a的排列。

既然我们已经知道怎么求三个字符的排列，那么固定第一个字符之后求后面两个字符的排列，就是典型的递归思路了。

基于前面的分析，我们可以得到如下的参考代码：

void Permutation(char\* pStr, char\* pBegin);

/////////////////////////////////////////////////////////////////////////

// Get the permutation of a string,

// for example, input string abc, its permutation is

// abc acb bac bca cba cab

/////////////////////////////////////////////////////////////////////////

void Permutation(char\* pStr)

{

Permutation(pStr, pStr);

}

/////////////////////////////////////////////////////////////////////////

// Print the permutation of a string,

// Input: pStr - input string

// pBegin - points to the begin char of string

// which we want to permutate in this recursion

/////////////////////////////////////////////////////////////////////////

void Permutation(char\* pStr, char\* pBegin)

{

if(!pStr || !pBegin)

return;

// if pBegin points to the end of string,

// this round of permutation is finished,

// print the permuted string

if(\*pBegin == '\0')

{

printf("%s\n", pStr);

}

// otherwise, permute string

else

{

for(char\* pCh = pBegin; \*pCh != '\0'; ++ pCh)

{

// swap pCh and pBegin

char temp = \*pCh;

\*pCh = \*pBegin;

\*pBegin = temp;

Permutation(pStr, pBegin + 1);

// restore pCh and pBegin

temp = \*pCh;

\*pCh = \*pBegin;

\*pBegin = temp;

}

}

}

扩展1：如果不是求字符的所有排列，而是求字符的所有组合，应该怎么办呢？当输入的字符串中含有相同的字符串时，相同的字符交换位置是不同的排列，但是同一个组合。举个例子，如果输入aaa，那么它的排列是6个aaa，但对应的组合只有一个。

扩展2：输入一个含有8个数字的数组，判断有没有可能把这8个数字分别放到正方体的8个顶点上，使得正方体上三组相对的面上的4个顶点的和相等。

# (59)-字符串的组合

题目：输入一个字符串，输出该字符串中字符的所有组合。举个例子，如果输入abc，它的组合有a、b、c、ab、ac、bc、abc。

分析：在本系列博客的第28题《[字符串的排列](http://zhedahht.blog.163.com/blog/static/254111742007499363479/)》中，我们详细讨论了如何用递归的思路求字符串的排列。同样，本题也可以用递归的思路来求字符串的组合。

假设我们想在长度为n的字符串中求m个字符的组合。我们先从头扫描字符串的第一个字符。针对第一个字符，我们有两种选择：一是把这个字符放到组合中去，接下来我们需要在剩下的n-1个字符中选取m-1个字符；而是不把这个字符放到组合中去，接下来我们需要在剩下的n-1个字符中选择m个字符。这两种选择都很容易用递归实现。下面是这种思路的参考代码：

void Combination(char\* string)

{

    if(string == NULL)

        return;

    int length = strlen(string);

    vector<char> result;

    for(int i = 1; i <= length; ++ i)

    {

        Combination(string, i, result);

    }

}

void Combination(char\* string, int number, vector<char>& result)

{

    if(number == 0)

    {

        vector<char>::iterator iter = result.begin();

        for(; iter < result.end(); ++ iter)

            printf("%c", \*iter);

        printf("\n");

        return;

    }

    if(\*string == '\0')

        return;

    result.push\_back(\*string);

    Combination(string + 1, number - 1, result);

    result.pop\_back();

    Combination(string + 1, number, result);

}

                由于组合可以是1个字符的组合，2个字符的字符……一直到n个字符的组合，因此在函数void Combination(char\* string)，我们需要一个for循环。另外，我们一个vector来存放选择放进组合里的字符。

# (20)－最长公共子串

题目：如果字符串一的所有字符按其在字符串中的顺序出现在另外一个字符串二中，则字符串一称之为字符串二的子串。注意，并不要求子串（字符串一）的字符必须连续出现在字符串二中。请编写一个函数，输入两个字符串，求它们的最长公共子串，并打印出最长公共子串。 例如：输入两个字符串BDCABA和ABCBDAB，字符串BCBA和BDAB都是是它们的最长公共子串，则输出它们的长度4，并打印任意一个子串。

分析：求最长公共子串（Longest Common Subsequence, LCS）是一道非常经典的动态规划题，因此一些重视算法的公司像MicroStrategy都把它当作面试题。

完整介绍动态规划将需要很长的篇幅，因此我不打算在此全面讨论动态规划相关的概念，只集中对LCS直接相关内容作讨论。如果对动态规划不是很熟悉，请参考相关算法书比如算法讨论。

先介绍LCS问题的性质：记Xm={x0, x1,…xm-1}和Yn={y0,y1,…,yn-1}为两个字符串，而Zk={z0,z1,…zk-1}是它们的LCS，则：

1. 如果xm-1=yn-1，那么zk-1=xm-1=yn-1，并且Zk-1是Xm-1和Yn-1的LCS；

2. 如果xm-1≠yn-1，那么当zk-1≠xm-1时Z是Xm-1和Y的LCS；

3. 如果xm-1≠yn-1，那么当zk-1≠yn-1时Z是Yn-1和X的LCS；

下面简单证明一下这些性质：

1. 如果zk-1≠xm-1，那么我们可以把xm-1（yn-1）加到Z中得到Z’，这样就得到X和Y的一个长度为k+1的公共子串Z’。这就与长度为k的Z是X和Y的LCS相矛盾了。因此一定有zk-1=xm-1=yn-1。

既然zk-1=xm-1=yn-1，那如果我们删除zk-1（xm-1、yn-1）得到的Zk-1，Xm-1和Yn-1，显然Zk-1是Xm-1和Yn-1的一个公共子串，现在我们证明Zk-1是Xm-1和Yn-1的LCS。用反证法不难证明。假设有Xm-1和Yn-1有一个长度超过k-1的公共子串W，那么我们把加到W中得到W’，那W’就是X和Y的公共子串，并且长度超过k，这就和已知条件相矛盾了。

2. 还是用反证法证明。假设Z不是Xm-1和Y的LCS，则存在一个长度超过k的W是Xm-1和Y的LCS，那W肯定也X和Y的公共子串，而已知条件中X和Y的公共子串的最大长度为k。矛盾。

3. 证明同2。

有了上面的性质，我们可以得出如下的思路：求两字符串Xm={x0, x1,…xm-1}和Yn={y0,y1,…,yn-1}的LCS，如果xm-1=yn-1，那么只需求得Xm-1和Yn-1的LCS，并在其后添加xm-1（yn-1）即可；如果xm-1≠yn-1，我们分别求得Xm-1和Y的LCS和Yn-1和X的LCS，并且这两个LCS中较长的一个为X和Y的LCS。

如果我们记字符串Xi和Yj的LCS的长度为c[i,j]，我们可以递归地求c[i,j]：

/ 0 if i<0 or j<0

c[i,j]= c[i-1,j-1]+1 if i,j>=0 and xi=xj

\ max(c[i,j-1],c[i-1,j] if i,j>=0 and xi≠xj

上面的公式用递归函数不难求得。但从前面求Fibonacci第n项(本面试题系列第16题）的分析中我们知道直接递归会有很多重复计算，我们用从底向上循环求解的思路效率更高。

为了能够采用循环求解的思路，我们用一个矩阵（参考代码中的LCS\_length）保存下来当前已经计算好了的c[i,j]，当后面的计算需要这些数据时就可以直接从矩阵读取。另外，求取c[i,j]可以从c[i-1,j-1] 、c[i,j-1]或者c[i-1,j]三个方向计算得到，相当于在矩阵LCS\_length中是从c[i-1,j-1]，c[i,j-1]或者c[i-1,j]的某一个各自移动到c[i,j]，因此在矩阵中有三种不同的移动方向：向左、向上和向左上方，其中只有向左上方移动时才表明找到LCS中的一个字符。于是我们需要用另外一个矩阵（参考代码中的LCS\_direction）保存移动的方向。

参考代码如下：

#include "string.h"

// directions of LCS generation

enum decreaseDir {kInit = 0, kLeft, kUp, kLeftUp};

/////////////////////////////////////////////////////////////////////////////

// Get the length of two strings' LCSs, and print one of the LCSs

// Input: pStr1 - the first string

// pStr2 - the second string

// Output: the length of two strings' LCSs

/////////////////////////////////////////////////////////////////////////////

int LCS(char\* pStr1, char\* pStr2)

{

if(!pStr1 || !pStr2)

return 0;

size\_t length1 = strlen(pStr1);

size\_t length2 = strlen(pStr2);

if(!length1 || !length2)

return 0;

size\_t i, j;

// initiate the length matrix

int \*\*LCS\_length;

LCS\_length = (int\*\*)(new int[length1]);

for(i = 0; i < length1; ++ i)

LCS\_length = (int\*)new int[length2];

for(i = 0; i < length1; ++ i)

for(j = 0; j < length2; ++ j)

LCS\_length[j] = 0;

// initiate the direction matrix

int \*\*LCS\_direction;

LCS\_direction = (int\*\*)(new int[length1]);

for( i = 0; i < length1; ++ i)

LCS\_direction = (int\*)new int[length2];

for(i = 0; i < length1; ++ i)

for(j = 0; j < length2; ++ j)

LCS\_direction[j] = kInit;

for(i = 0; i < length1; ++ i)

{

for(j = 0; j < length2; ++ j)

{

if(i == 0 || j == 0)

{

if(pStr1 == pStr2[j])

{

LCS\_length[j] = 1;

LCS\_direction[j] = kLeftUp;

}

else

LCS\_length[j] = 0;

}

// a char of LCS is found,

// it comes from the left up entry in the direction matrix

else if(pStr1 == pStr2[j])

{

LCS\_length[j] = LCS\_length[i - 1][j - 1] + 1;

LCS\_direction[j] = kLeftUp;

}

// it comes from the up entry in the direction matrix

else if(LCS\_length[i - 1][j] > LCS\_length[j - 1])

{

LCS\_length[j] = LCS\_length[i - 1][j];

LCS\_direction[j] = kUp;

}

// it comes from the left entry in the direction matrix

else

{

LCS\_length[j] = LCS\_length[j - 1];

LCS\_direction[j] = kLeft;

}

}

}

LCS\_Print(LCS\_direction, pStr1, pStr2, length1 - 1, length2 - 1);

return LCS\_length[length1 - 1][length2 - 1];

}

/////////////////////////////////////////////////////////////////////////////

// Print a LCS for two strings

// Input: LCS\_direction - a 2d matrix which records the direction of

// LCS generation

// pStr1 - the first string

// pStr2 - the second string

// row - the row index in the matrix LCS\_direction

// col - the column index in the matrix LCS\_direction

/////////////////////////////////////////////////////////////////////////////

void LCS\_Print(int \*\*LCS\_direction,

char\* pStr1, char\* pStr2,

size\_t row, size\_t col)

{

if(pStr1 == NULL || pStr2 == NULL)

return;

size\_t length1 = strlen(pStr1);

size\_t length2 = strlen(pStr2);

if(length1 == 0 || length2 == 0 || !(row < length1 && col < length2))

return;

// kLeftUp implies a char in the LCS is found

if(LCS\_direction[row][col] == kLeftUp)

{

if(row > 0 && col > 0)

LCS\_Print(LCS\_direction, pStr1, pStr2, row - 1, col - 1);

// print the char

printf("%c", pStr1[row]);

}

else if(LCS\_direction[row][col] == kLeft)

{

// move to the left entry in the direction matrix

if(col > 0)

LCS\_Print(LCS\_direction, pStr1, pStr2, row, col - 1);

}

else if(LCS\_direction[row][col] == kUp)

{

// move to the up entry in the direction matrix

if(row > 0)

LCS\_Print(LCS\_direction, pStr1, pStr2, row - 1, col);

}

}

扩展：如果题目改成求两个字符串的最长公共子字符串，应该怎么求？子字符串的定义和子串的定义类似，但要求是连续分布在其他字符串中。比如输入两个字符串BDCABA和ABCBDAB的最长公共字符串有BD和AB，它们的长度都是2。