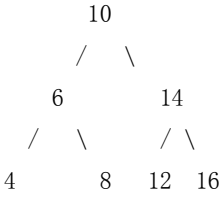
http://zhedahht.blog.163.com/blog/static/254111742007127104759245/

程序员面试题精选 100 题

(01)－把二元查找树转变成排序的双向链表

题目：输入一棵二元查找树，将该二元查找树转换成一个排序的双向链表。要求不能创建任何新的结点，只调整指针的指向。

比如将二元查找树



转换成双向链表

4=6=8=10=12=14=16。

分析：本题是微软的面试题。很多与树相关的题目都是用递归的思路来解决，本题也不例外。下面我们用两种不同的递归思路来分析。

思路一：当我们到达某一结点准备调整以该结点为根结点的子树时，先调整其左子树将左子树转换成一个排好序的左子链表，再调整其右子树转换右子链表。最近链接左子链表的最右结点（左子树的最大结点）、当前结点和右子链表的最左结点（右子树的最小结点）。从树的根结点开始递归调整所有结点。

思路二：我们可以中序遍历整棵树。按照这个方式遍历树，比较小的结点先访问。如果我们每访问一个结点，假设之前访问过的结点已经调整成一个排序双向链表，我们再把调整当前结点的指针将其链接到链表的末尾。当所有结点都访问过之后，整棵树也就转换成一个排序双向链表了。

参考代码：

首先我们定义二元查找树结点的数据结构如下：

```
struct BSTreeNode // a node in the binary search tree
{
 int         m_nValue; // value of node
BSTreeNode  *m_pLeft;  // left child of node
BSTreeNode  *m_pRight; // right child of node
};
```

思路一对应的代码：

```
//////////////////////////////////////////////////////////////////////
//
// Covert a sub binary-search-tree into a sorted double-linked list
// Input: pNode - the head of the sub tree
//       asRight - whether pNode is the right child of its parent
// Output: if asRight is true, return the least node in the sub-tree
//        else return the greatest node in the sub-tree
//////////////////////////////////////////////////////////////////////
//
BSTreeNode* ConvertNode(BSTreeNode* pNode, bool asRight)
{
    if(!pNode)
        return NULL;

    BSTreeNode *pLeft = NULL;
    BSTreeNode *pRight = NULL;

    // Convert the left sub-tree
    if(pNode->m_pLeft)
        pLeft = ConvertNode(pNode->m_pLeft, false);

    // Connect the greatest node in the left sub-tree to the current node
    if(pLeft)
    {
        pLeft->m_pRight = pNode;
        pNode->m_pLeft = pLeft;
    }

    // Convert the right sub-tree
    if(pNode->m_pRight)
        pRight = ConvertNode(pNode->m_pRight, true);

    // Connect the least node in the right sub-tree to the current node
    if(pRight)
    {
        pNode->m_pRight = pRight;
        pRight->m_pLeft = pNode;
    }

    BSTreeNode *pTemp = pNode;

    // If the current node is the right child of its parent,
    // return the least node in the tree whose root is the current node
```

```
    if(asRight)
    {
        while(pTemp->m_pLeft)
            pTemp = pTemp->m_pLeft;
    }
    // If the current node is the left child of its parent,
    // return the greatest node in the tree whose root is the current node
    else
    {
        while(pTemp->m_pRight)
            pTemp = pTemp->m_pRight;
    }

    return pTemp;
}

//////////////////////////////////////////////////////////////////////
//
// Covert a binary search tree into a sorted double-linked list
// Input: the head of tree
// Output: the head of sorted double-linked list
//////////////////////////////////////////////////////////////////////
//
BSTreeNode* Convert(BSTreeNode* pHeadOfTree)
{
    // As we want to return the head of the sorted double-linked list,
    // we set the second parameter to be true
    return ConvertNode(pHeadOfTree, true);
}
```

思路二对应的代码：

```
//////////////////////////////////////////////////////////////////////
//
// Covert a sub binary-search-tree into a sorted double-linked list
// Input: pNode -          the head of the sub tree
//       pLastNodeInList - the tail of the double-linked list
//////////////////////////////////////////////////////////////////////
//
void ConvertNode(BSTreeNode* pNode, BSTreeNode*& pLastNodeInList)
{
    if(pNode == NULL)
        return;

    BSTreeNode *pCurrent = pNode;
```

```
    // Convert the left sub-tree
    if (pCurrent->m_pLeft != NULL)
        ConvertNode(pCurrent->m_pLeft, pLastNodeInList);

    // Put the current node into the double-linked list
    pCurrent->m_pLeft = pLastNodeInList;
    if(pLastNodeInList != NULL)
        pLastNodeInList->m_pRight = pCurrent;

    pLastNodeInList = pCurrent;

    // Convert the right sub-tree
    if (pCurrent->m_pRight != NULL)
        ConvertNode(pCurrent->m_pRight, pLastNodeInList);
}

//////////////////////////////////////////////////////////////////////
//
// Covert a binary search tree into a sorted double-linked list
// Input: pHeadOfTree - the head of tree
// Output: the head of sorted double-linked list
//////////////////////////////////////////////////////////////////////
//
BSTreeNode* Convert_Solution1(BSTreeNode* pHeadOfTree)
{
    BSTreeNode *pLastNodeInList = NULL;
    ConvertNode(pHeadOfTree, pLastNodeInList);

    // Get the head of the double-linked list
    BSTreeNode *pHeadOfList = pLastNodeInList;
    while(pHeadOfList && pHeadOfList->m_pLeft)
        pHeadOfList = pHeadOfList->m_pLeft;

    return pHeadOfList;
}
```

(02)－设计包含 min 函数的栈

题目：定义栈的数据结构，要求添加一个 min 函数，能够得到栈的最小元素。要求函数 min、push 以及 pop 的时间复杂度都是 O(1)。

分析：这是去年 google 的一道面试题。

我看到这道题目时，第一反应就是每次 push 一个新元素时，将栈里所有逆序元素排序。这样栈顶元素将是最小元素。但由于不能保证最后 push 进栈的元素最先出栈，这种思路设计的数据结构已经不是一个栈了。

在栈里添加一个成员变量存放最小元素（或最小元素的位置）。每次 push 一个新元素进栈的时候，如果该元素比当前的最小元素还要小，则更新最小元素。

乍一看这样思路挺好的。但仔细一想，该思路存在一个重要的问题：如果当前最小元素被 pop 出去，如何才能得到下一个最小元素？

因此仅仅只添加一个成员变量存放最小元素（或最小元素的位置）是不够的。我们需要一个辅助栈。每次 push 一个新元素的时候，同时将最小元素（或最小元素的位置。考虑到栈元素的类型可能是复杂的数据结构，用最小元素的位置将能减少空间消耗）push 到辅助栈中；每次 pop 一个元素出栈的时候，同时 pop 辅助栈。

参考代码：

```
#include <deque>
#include <assert.h>

template <typename T> class CStackWithMin
{
public:
    CStackWithMin(void) {}
    virtual ~CStackWithMin(void) {}

    T& top(void);
    const T& top(void) const;

    void push(const T& value);
    void pop(void);

    const T& min(void) const;

private:
    T>m_data;// theelements of stack
    size_t>m_minIndex;// the indicesof minimum elements
```

```
};

// get the last element of mutable stack
template <typename T> T& CStackWithMin<T>::top()
{
    return m_data.back();
}

// get the last element of non-mutable stack
template <typename T> const T& CStackWithMin<T>::top() const
{
    return m_data.back();
}

// insert an elment at the end of stack
template <typename T> void CStackWithMin<T>::push(const T& value)
{
    // append the data into the end of m_data
    m_data.push_back(value);

    // set the index of minimum elment in m_data at the end of m_minIndex
    if(m_minIndex.size() == 0)
        m_minIndex.push_back(0);
    else
    {
        if(value < m_data[m_minIndex.back()])
            m_minIndex.push_back(m_data.size() - 1);
        else
            m_minIndex.push_back(m_minIndex.back());
    }
}

// erease the element at the end of stack
template <typename T> void CStackWithMin<T>::pop()
{
    // pop m_data
    m_data.pop_back();

    // pop m_minIndex
    m_minIndex.pop_back();
}

// get the minimum element of stack
template <typename T> const T& CStackWithMin<T>::min() const
```

```
{
    assert(m_data.size() > 0);
    assert(m_minIndex.size() > 0);

    return m_data[m_minIndex.back()];
}
```

举个例子演示上述代码的运行过程：

步骤	数据栈	辅助栈	最小值
1.push 3	3	0	3
2.push 4	3,4	0,0	3
3.push 2	3,4,2	0,0,2	2
4.push 1	3,4,2,1	0,0,2,3	1
5.pop	3,4,2	0,0,2	2
6.pop	3,4	0,0	3
7.push 0	3,4,0	0,0,2	0

讨论：如果思路正确，编写上述代码不是一件很难的事情。但如果能注意一些细节无疑能在面试中加分。比如我在上面的代码中做了如下的工作：

- 用模板类实现。如果别人的元素类型只是 int 类型，模板将能给面试官带来好印象；
- 两个版本的 top 函数。在很多类中，都需要提供 const 和非 const 版本的成员访问函数；
- min 函数中 assert。把代码写的尽量安全是每个软件公司对程序员的要求；
- 添加一些注释。注释既能提高代码的可读性，又能增加代码量，何乐而不为？

总之，在面试时如果时间允许，尽量把代码写的漂亮一些。说不定代码中的几个小亮点就能让自己轻松拿到心仪的 Offer。

PS：每当 push 进一个新元素，若比当前最小元素小，则将它进栈，并将它的 index 进最小辅助栈；若大于当前最小元素，则将它进栈，并将当前最小元素 index 进最小辅助栈（可以重复进栈多次）！

---

(03)一求子数组的最大和

题目：输入一个整形数组，数组里有正数也有负数。数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。求所有子数组的和的最大值。要求时间复杂度为 O(n)。

例如输入的数组为 1, -2, 3, 10, -4, 7, 2, -5，和最大的子数组为 3, 10, -4, 7, 2，因此输出为该子数组的和 18。

分析：本题最初为 2005 年浙江大学计算机系的考研题的最后一道程序设计题，在 2006 年里包括 google 在内的很多知名公司都把本题当作面试题。由于本题在网络中广为流传，本题也顺利成为 2006 年程序员面试题中经典中的经典。

如果不考虑时间复杂度，我们可以枚举出所有子数组并求出他们的和。不过非常遗憾的是，由于长度为 n 的数组有 $O(n^2)$个子数组；而且求一个长度为 n 的数组的和的时间复杂度为 O(n)。因此这种思路的时间是 $O(n^3)$。

很容易理解，当我们加上一个正数时，和会增加；当我们加上一个负数时，和会减少。如果当前得到的和是个负数，那么这个和在接下来的累加中应该抛弃并重新清零，不然的话这个负数将会减少接下来的和。基于这样的思路，我们可以写出如下代码。

参考代码：

```
//////////////////////////////////////////////////////////////////////
////////
// Find the greatest sum of all sub-arrays
// Return value: if the input is valid, return true, otherwise return false
//////////////////////////////////////////////////////////////////////
////////
bool FindGreatestSumOfSubArray
(
    int *pData,          // an array
    unsigned int nLength, // the length of array
    int &nGreatestSum     // the greatest sum of all sub-arrays
)
{
    // if the input is invalid, return false
    if((pData == NULL) || (nLength == 0))
        return false;

    int nCurSum = nGreatestSum = 0;
    for(unsigned int i = 0; i < nLength; ++i)
    {
        nCurSum += pData[i];

        // if the current sum is negative, discard it
        if(nCurSum < 0)
            nCurSum = 0;

        // if a greater sum is found, update the greatest sum
        if(nCurSum > nGreatestSum)
            nGreatestSum = nCurSum;
```

```
    }


    // if all data are negative, find the greatest element in the array
    if(nGreatestSum == 0)
    {
        nGreatestSum = pData[0];
        for(unsigned int i = 1; i < nLength; ++i)
        {
            if(pData[i] > nGreatestSum)
                nGreatestSum = pData[i];
        }
    }

    return true;
}
```

讨论：上述代码中有两点值得和大家讨论一下：

- 函数的返回值不是子数组和的最大值，而是一个判断输入是否有效的标志。如果函数返回值的是子数组和的最大值，那么当输入一个空指针是应该返回什么呢？返回 0？那这个函数的用户怎么区分输入无效和子数组和的最大值刚好是 0 这两中情况呢？基于这个考虑，本人认为把子数组和的最大值以引用的方式放到参数列表中，同时让函数返回一个函数是否正常执行的标志。

- 输入有一类特殊情况需要特殊处理。当输入数组中所有整数都是负数时，子数组和的最大值就是数组中的最大元素。

《编程珠机》第八章，8.4 扫描算法。

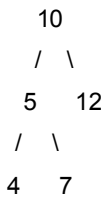
采用类似分治算法的道理：前 i 个元素中，最大综合子数组要么在 i-1 个元素中(maxsofar)，要么截止到位置 i(maxendinghere)。

---

(04)一在二元树中找出和为某一值的所有路径

题目：输入一个整数和一棵二元树。从树的根结点开始往下访问一直到叶结点所经过的所有结点形成一条路径。打印出和与输入整数相等的所有路径。

例如输入整数 22 和如下二元树



则打印出两条路径：10, 12 和 10, 5, 7。

二元树结点的数据结构定义为：

```
struct BinaryTreeNode // a node in the binary tree
{
    int            m_nValue; // value of node
    BinaryTreeNode  *m_pLeft;  // left child of node
    BinaryTreeNode  *m_pRight; // right child of node
};
```

分析：这是百度的一道笔试题，考查对树这种基本数据结构以及递归函数的理解。

当访问到某一结点时，把该结点添加到路径上，并累加当前结点的值。如果当前结点为叶结点并且当前路径的和刚好等于输入的整数，则当前的路径符合要求，我们把它打印出来。如果当前结点不是叶结点，则继续访问它的子结点。当前结点访问结束后，递归函数将自动回到父结点。因此我们在函数退出之前要在路径上删除当前结点并减去当前结点的值，以确保返回父结点时路径刚好是根结点到父结点的路径。我们不难看出保存路径的数据结构实际上是一个栈结构，因为路径要与递归调用状态一致，而递归调用本质就是一个压栈和出栈的过程。

参考代码：

```
///////////////////////////////////////////////////////////////////////
//
// Find paths whose sum equal to expected sum
///////////////////////////////////////////////////////////////////////
//
void FindPath
(
    BinaryTreeNode*   pTreeNode,    // a node of binary tree
    int             expectedSum,  // the expected sum
    std::vector<int>&path,       // a pathfrom root to current node
    int&            currentSum    // the sum of path
)
{
    if(!pTreeNode)
        return;
```

```
    currentSum += pTreeNode->m_nValue;
    path.push_back(pTreeNode->m_nValue);

    // if the node is a leaf, and the sum is same as pre-defined,
    // the path is what we want. print the path
    bool isLeaf = (!pTreeNode->m_pLeft && !pTreeNode->m_pRight);
    if(currentSum == expectedSum && isLeaf)
    {
std::vector<int>::iterator iter =path.begin();
for(; iter != path.end(); ++ iter)
           std::cout<<*iter<<'\t';
       std::cout<<std::endl;
    }

    // if the node is not a leaf, goto its children
    if(pTreeNode->m_pLeft)
        FindPath(pTreeNode->m_pLeft, expectedSum, path, currentSum);
    if(pTreeNode->m_pRight)
        FindPath(pTreeNode->m_pRight, expectedSum, path, currentSum);

    // when we finish visiting a node and return to its parent node,
    // we should delete this node from the path and
    // minus the node's value from the current sum
    currentSum -= pTreeNode->m_nValue;
    path.pop_back();
}
```

---

(05)一查找最小的 k 个元素

题目：输入 n 个整数，输出其中最小的 k 个。

例如输入 1，2，3，4，5，6，7 和 8 这 8 个数字，则最小的 4 个数字为 1，2，3 和 4。

分析：这道题最简单的思路莫过于把输入的 n 个整数排序，这样排在最前面的 k 个数就是最小的 k 个数。只是这种思路的时间复杂度为 O(n*log*n)。我们试着寻找更快的解决思路。

我们可以开辟一个长度为 k 的数组。每次从输入的 n 个整数中读入一个数。如果数组中已经插入的元素少于 k 个，则将读入的整数直接放到数组中。否则长度为 k 的数组已经满了，不能再往数组里插入元素，只

能替换了。如果读入的这个整数比数组中已有 k 个整数的最大值要小，则用读入的这个整数替换这个最大值；如果读入的整数比数组中已有 k 个整数的最大值还要大，则读入的这个整数不可能是最小的 k 个整数之一，抛弃这个整数。这种思路相当于只要排序 k 个整数，因此时间复杂可以降到 O(n+n*log*k)。通常情况下 k 要远小于 n，所以这种办法要优于前面的思路。

这是我能够想出来的最快的解决方案。不过从给面试官留下更好印象的角度出发，我们可以进一步把代码写得更漂亮一些。从上面的分析，当长度为 k 的数组已经满了之后，如果需要替换，每次替换的都是数组中的最大值。在常用的数据结构中，能够在 O(1)时间里得到最大值的数据结构为最大堆。因此我们可以用堆（heap）来代替数组。

另外，自己重头开始写一个最大堆需要一定量的代码。我们现在不需要重新去发明车轮，因为前人早就发明出来了。同样，STL 中的 set 和 multiset 为我们做了很好的堆的实现，我们可以拿过来用。既偷了懒，又给面试官留下熟悉 STL 的好印象，何乐而不为之？

参考代码：

```
#include <set>
#include <vector>
#include <iostream>

using namespace std;

typedef multiset<int, greater<int> >  IntHeap;

//////////////////////////////////////////////////////////////////////
//
// find k least numbers in a vector
//////////////////////////////////////////////////////////////////////
//
void FindKLeastNumbers
(
    const vector<int>& data,              // a vector of data
    IntHeap& leastNumbers,                // k least numbers, output
    unsigned int k
)
{
    leastNumbers.clear();

    if(k == 0 || data.size() < k)
        return;

    vector<int>::const_iterator iter = data.begin();
    for(; iter != data.end(); ++ iter)
    {
        // if less than k numbers was inserted into leastNumbers
```

```
        if((leastNumbers.size()) < k)
            leastNumbers.insert(*iter);

        // leastNumbers contains k numbers and it's full now
        else
        {
            // first number in leastNumbers is the greatest one
            IntHeap::iterator iterFirst = leastNumbers.begin();

            // if is less than the previous greatest number
            if(*iter < *(leastNumbers.begin()))
            {
                // replace the previous greatest number
                leastNumbers.erase(iterFirst);
                leastNumbers.insert(*iter);
            }
        }
    }
}
```

---

(06)一判断整数序列是不是二元查找树的后序遍历结果

题目：输入一个整数数组，判断该数组是不是某二元查找树的后序遍历的结果。如果是返回 true，否则返回 false。

例如输入 5、7、6、9、11、10、8，由于这一整数序列是如下树的后序遍历结果：

```
        8
      /  \
     6    10
   /\    /\
  5  7  9  11
```

因此返回 true。

如果输入 7、4、6、5，没有哪棵树的后序遍历的结果是这个序列，因此返回 false。

分析：这是一道 trilogy 的笔试题，主要考查对二元查找树的理解。

在后续遍历得到的序列中，最后一个元素为树的根结点。从头开始扫描这个序列，比根结点小的元素都应该位于序列的左半部分；从第一个大于跟结点开始到跟结点前面的一个元素为止，所有元素都应该大于跟结点，因为这部分元素对应的是树的右子树。根据这样的划分，把序列划分为左右两部分，我们递归地确认序列的左、右两部分是不是都是二元查找树。

参考代码：

```
using namespace std;

////////////////////////////////////////////////////////////////////////
//
// Verify whether a squence of integers are the post order traversal
// of a binary search tree (BST)
// Input: squence - the squence of integers
//        length  - the length of squence
// Return: return ture if the squence is traversal result of a BST,
//         otherwise, return false
////////////////////////////////////////////////////////////////////////
//
bool verifySquenceOfBST(int squence[], int length)
{
    if(squence == NULL || length <= 0)
        return false;

    // root of a BST is at the end of post order traversal squence
    int root = squence[length - 1];

    // the nodes in left sub-tree are less than the root
    int i = 0;
    for(; i < length - 1; ++ i)
    {
        if(squence[i] > root)
            break;
    }

    // the nodes in the right sub-tree are greater than the root
    int j = i;
    for(; j < length - 1; ++ j)
    {
        if(squence[j] < root)
            return false;
    }

    // verify whether the left sub-tree is a BST
    bool left = true;
```

```
    if(i > 0)
        left = verifySquenceOfBST(squence, i);

    // verify whether the right sub-tree is a BST
    bool right = true;
    if(i < length - 1)
        right = verifySquenceOfBST(squence + i, length - i - 1);

    return (left && right);
}
```

---

(07)一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;
}
```

---

(08)一求 1+2+...+n

题目：求 1+2+…+n，要求不能使用乘除法、for、while、if、else、switch、case 等关键字以及条件判断语句（A?B:C）。

分析：这道题没有多少实际意义，因为在软件开发中不会有这么变态的限制。但这道题却能有效地考查发散思维能力，而发散思维能力能反映出对编程相关技术理解的深刻程度。

通常求 1+2+…+n 除了用公式 n(n+1)/2 之外，无外乎循环和递归两种思路。由于已经明确限制 for 和 while 的使用，循环已经不能再用了。同样，递归函数也需要用 if 语句或者条件判断语句来判断是继续递归下去还是终止递归，但现在题目已经不允许使用这两种语句了。

我们仍然围绕循环做文章。循环只是让相同的代码执行 n 遍而已，我们完全可以不用 for 和 while 达到这个效果。比如定义一个类，我们 new 一含有 n 个这种类型元素的数组，那么该类的构造函数将确定会被调用 n 次。我们可以将需要执行的代码放到构造函数里。如下代码正是基于这个思路：

```
class Temp
{
public:
    Temp() { ++ N; Sum += N; }

    static void Reset() { N = 0; Sum = 0; }
    static int GetSum() { return Sum; }

private:
```

```
    static int N;
    static int Sum;
};

int Temp::N = 0;
int Temp::Sum = 0;

int solution1_Sum(int n)
{
      Temp::Reset();

    Temp *a = new Temp[n];
    delete []a;
    a = 0;

      return Temp::GetSum();
}
```

我们同样也可以围绕递归做文章。既然不能判断是不是应该终止递归，我们不妨定义两个函数。一个函数充当递归函数的角色，另一个函数处理终止递归的情况，我们需要做的就是在两个函数里二选一。从二选一我们很自然的想到布尔变量，比如 ture（1）的时候调用第一个函数，false（0）的时候调用第二个函数。那现在的问题是如和把数值变量 n 转换成布尔值。如果对 n 连续做两次反运算，即!!n，那么非零的 n 转换为 true，0 转换为 false。有了上述分析，我们再来看下面的代码：

```
class A;
A* Array[2];

class A
{
public:
    virtual int Sum (int n) { return 0; }
};

class B: public A
{
public:
    virtual int Sum (int n) { return Array[!!n]->Sum(n-1)+n; }
};

int solution2_Sum(int n)
{
    A a;
    B b;
    Array[0] = &a;
    Array[1] = &b;
```

```
    int value = Array[1]->Sum(n);

    return value;
}
```

这种方法是用虚函数来实现函数的选择。当 n 不为零时，执行函数 B::Sum；当 n 为 0 时，执行 A::Sum。我们也可以直接用函数指针数组，这样可能还更直接一些：

```
typedef int (*fun)(int);

int solution3_f1(int i)
{
    return 0;
}

int solution3_f2(int i)
{
    fun f[2]={solution3_f1, solution3_f2};
    return i+f[!!i](i-1);
}
```

另外我们还可以让编译器帮我们来完成类似于递归的运算，比如如下代码：

```
template <int n> struct solution4_Sum
{
    enum Value { N = solution4_Sum<n - 1>::N + n};
};

template <> struct solution4_Sum<1>
{
    enum Value { N = 1};
};
```

solution4_Sum<100>::N 就是 1+2+...+100 的结果。当编译器看到 solution4_Sum<100>时，就是为模板类 solution4_Sum 以参数 100 生成该类型的代码。但以 100 为参数的类型需要得到以 99 为参数的类型，因为 solution4_Sum<100>::N=solution4_Sum<99>::N+100。这个过程会递归一直到参数为 1 的类型，由于该类型已经显式定义，编译器无需生成，递归编译到此结束。由于这个过程是在编译过程中完成的，因此要求输入 n 必须是在编译期间就能确定，不能动态输入。这是该方法最大的缺点。而且编译器对递归编译代码的递归深度是有限制的，也就是要求 n 不能太大。

大家还有更多、更巧妙的思路吗？欢迎讨论^_^

PS:递归解决

```
int func(int n)
{
        int i=1;
        (n>1)&&(i=func(n-1)+n);
        return i;
}
```

---

(09)－查找链表中倒数第 k 个结点

题目：输入一个单向链表，输出该链表中倒数第 k 个结点。链表的倒数第 0 个结点为链表的尾指针。链表结点定义如下：

```
struct ListNode
{
    int      m_nKey;
    ListNode* m_pNext;
};
```

分析：为了得到倒数第 k 个结点，很自然的想法是先走到链表的尾端，再从尾端回溯 k 步。可是输入的是单向链表，只有从前往后的指针而没有从后往前的指针。因此我们需要打开我们的思路。

既然不能从尾结点开始遍历这个链表，我们还是把思路回到头结点上来。假设整个链表有 n 个结点，那么倒数第 k 个结点是从头结点开始的第 n-k-1 个结点（从 0 开始计数）。如果我们能够得到链表中结点的个数 n，那我们只要从头结点开始往后走 n-k-1 步就可以了。如何得到结点数 n？这个不难，只需要从头开始遍历链表，每经过一个结点，计数器加一就行了。

这种思路的时间复杂度是 O(n)，但需要遍历链表两次。第一次得到链表中结点个数 n，第二次得到从头结点开始的第 n-k-1 个结点即倒数第 k 个结点。

如果链表的结点数不多，这是一种很好的方法。但如果输入的链表的结点个数很多，有可能不能一次性把整个链表都从硬盘读入物理内存，那么遍历两遍意味着一个结点需要两次从硬盘读入到物理内存。我们知道把数据从硬盘读入到内存是非常耗时间的操作。我们能不能把链表遍历的次数减少到 1？如果可以，将能有效地提高代码执行的时间效率。

如果我们在遍历时维持两个指针，第一个指针从链表的头指针开始遍历，在第 k-1 步之前，第二个指针保持不动；在第 k-1 步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在 k-1，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第 k 个结点。

这种思路只需要遍历链表一次。对于很长的链表，只需要把每个结点从硬盘导入到内存一次。因此这一方法的时间效率前面的方法要高。

思路一的参考代码：

```
////////////////////////////////////////////////////////////////////
//
// Find the kth node from the tail of a list
// Input: pListHead - the head of list
//        k         - the distance to the tail
// Output: the kth node from the tail of a list
////////////////////////////////////////////////////////////////////
//
ListNode* FindKthToTail_Solution1(ListNode* pListHead, unsigned int k)
{
    if(pListHead == NULL)
        return NULL;

    // count the nodes number in the list
    ListNode *pCur = pListHead;
    unsigned int nNum = 0;
    while(pCur->m_pNext != NULL)
    {
        pCur = pCur->m_pNext;
        nNum ++;
    }

    // if the number of nodes in the list is less than k
    // do nothing
    if(nNum < k)
        return NULL;

    // the kth node from the tail of a list
    // is the (n - k)th node from the head
    pCur = pListHead;
    for(unsigned int i = 0; i < nNum - k; ++ i)
        pCur = pCur->m_pNext;

    return pCur;
}
```

思路二的参考代码：

```
////////////////////////////////////////////////////////////////////
//
```

```
// Find the kth node from the tail of a list
// Input: pListHead - the head of list
//        k         - the distance to the tail
// Output: the kth node from the tail of a list
////////////////////////////////////////////////////////////////////
//
ListNode* FindKthToTail_Solution2(ListNode* pListHead, unsigned int k)
{
    if(pListHead == NULL)
        return NULL;

    ListNode *pAhead = pListHead;
    ListNode *pBehind = NULL;

    for(unsigned int i = 0; i < k; ++ i)
    {
        if(pAhead->m_pNext != NULL)
            pAhead = pAhead->m_pNext;
        else
        {
            // if the number of nodes in the list is less than k,
            // do nothing
            return NULL;
        }
    }

    pBehind = pListHead;

    // the distance between pAhead and pBehind is k
    // when pAhead arrives at the tail, p
    // Behind is at the kth node from the tail
    while(pAhead->m_pNext != NULL)
    {
        pAhead = pAhead->m_pNext;
        pBehind = pBehind->m_pNext;
    }

    return pBehind;
}
```

讨论：这道题的代码有大量的指针操作。在软件开发中，错误的指针操作是大部分问题的根源。因此每个公司都希望程序员在操作指针时有良好的习惯，比如使用指针之前判断是不是空指针。这些都是编程的细节，但如果这些细节把握得不好，很有可能就会和心仪的公司失之交臂。

另外，这两种思路对应的代码都含有循环。含有循环的代码经常出的问题是在循环结束条件的判断。是该用小于还是小于等于？是该用 k 还是该用 k-1？由于题目要求的是从 0 开始计数，而我们的习惯思维是从 1 开始计数，因此首先要想好这些边界条件再开始编写代码，再者要在编写完代码之后再用边界值、边界值减 1、边界值加 1 都运行一次（在纸上写代码就只能在心里运行了）。

扩展：和这道题类似的题目还有：输入一个单向链表。如果该链表的结点数为奇数，输出中间的结点；如果链表结点数为偶数，输出中间两个结点前面的一个。如果各位感兴趣，请自己分析并编写代码。

解扩展题的思路和思路二类似吧，也用到两个指针。节点数为奇数时，两个指针初始都指向头结点，然后一个每次跳两个节点，一个每次跳一个节点。当第一个指针到达尾端时后一个指针指向需要的节点。

节点数为偶数时情况基本一样，只是两个指针初始一个指向头结点，一个指向 1 号节点。后一个指针每次跳两个节点，前一个每次只跳一个节点。

---

(10)－在排序数组中查找和为给定值的两个数字

题目：输入一个已经按升序排序过的数组和一个数字，在数组中查找两个数，使得它们的和正好是输入的那个数字。要求时间复杂度是 O(n)。如果有多对数字的和等于输入的数字，输出任意一对即可。

例如输入数组 1、2、4、7、11、15 和数字 15。由于 4+11=15，因此输出 4 和 11。

分析：如果我们不考虑时间复杂度，最简单想法的莫过去先在数组中固定一个数字，再依次判断数组中剩下的 n-1 个数字与它的和是不是等于输入的数字。可惜这种思路需要的时间复杂度是 $O(n^2)$。

我们假设现在随便在数组中找到两个数。如果它们的和等于输入的数字，那太好了，我们找到了要找的两个数字；如果小于输入的数字呢？我们希望两个数字的和再大一点。由于数组已经排好序了，我们是不是可以把较小的数字的往后面移动一个数字？因为排在后面的数字要大一些，那么两个数字的和也要大一些，就有可能等于输入的数字了；同样，当两个数字的和大于输入的数字的时候，我们把较大的数字往前移动，因为排在数组前面的数字要小一些，它们的和就有可能等于输入的数字了。

我们把前面的思路整理一下：最初我们找到数组的第一个数字和最后一个数字。当两个数字的和大于输入的数字时，把较大的数字往前移动；当两个数字的和小于数字时，把较小的数字往后移动；当相等时，打完收工。这样扫描的顺序是从数组的两端向数组的中间扫描。

问题是这样的思路是不是正确的呢？这需要严格的数学证明。感兴趣的读者可以自行证明一下。

参考代码：

```
////////////////////////////////////////////////////////////////////////
//
// Find two numbers with a sum in a sorted array
// Output: ture is found such two numbers, otherwise false
////////////////////////////////////////////////////////////////////////
//
bool FindTwoNumbersWithSum
(
    int data[],          // a sorted array
    unsigned int length,  // the length of the sorted array
    int sum,             // the sum
    int& num1,           // the first number, output
    int& num2            // the second number, output
)
{

    bool found = false;
    if(length < 1)
        return found;

    int ahead = length - 1;
    int behind = 0;

    while(ahead > behind)
    {
        long long curSum = data[ahead] + data[behind];

        // if the sum of two numbers is equal to the input
        // we have found them
        if(curSum == sum)
        {
            num1 = data[behind];
            num2 = data[ahead];
            found = true;
            break;
        }
        // if the sum of two numbers is greater than the input
        // decrease the greater number
        else if(curSum > sum)
            ahead --;
        // if the sum of two numbers is less than the input
        // increase the less number
        else
            behind ++;
```

```
    }

    return found;
}
```

扩展：如果输入的数组是没有排序的，但知道里面数字的范围，其他条件不变，如和在O(n)时间里找到这两个数字？

==========

扩展问题是不是先记数排序再用原来的方法？

如果是这样的话，设数字范围是 d，则时间复杂度应该是 O(max(d,n))

是这个思路。如果记最小值为 min，最大值为 max。新建一个长度为 max-min+1 的数组。初始化这个数组的每个元素为 0。扫描原数组每个元素 k，在新数组中下标为 k-min 的位置加 1，这样在 O(n)的时间内把原数组转换为一个排好序的数组。接下来的做法一样。

当然，时间复杂度标记为 O(max(d,n)) 更准确一些。谢谢指出。

---

(11)－求二元查找树的镜像

题目：输入一颗二元查找树，将该树转换为它的镜像，即在转换后的二元查找树中，左子树的结点都大于右子树的结点。用递归和循环两种方法完成树的镜像转换。

例如输入：

```
      8
    /   \
   6     10
  /\     /\
5   7   9   11
```

输出：

```
      8
    /   \
```

```
  10    6
 /\    /\
11  9 7  5
```

定义二元查找树的结点为：

```
struct BSTreeNode // a node in the binary search tree (BST)
{
    int         m_nValue; // value of node
    BSTreeNode  *m_pLeft;  // left child of node
    BSTreeNode  *m_pRight; // right child of node
};
```

分析：尽管我们可能一下子不能理解镜像是什么意思，但上面的例子给我们的直观感觉，就是交换结点的左右子树。我们试着在遍历例子中的二元查找树的同时来交换每个结点的左右子树。遍历时首先访问头结点 8，我们交换它的左右子树得到：

```
     8
    / \
   10   6
  /\    /\
 9  11  5  7
```

我们发现两个结点 6 和 10 的左右子树仍然是左结点的值小于右结点的值，我们再试着交换他们的左右子树，得到：

```
    8
   / \
  10   6
  /\    /\
11  9  7  5
```

刚好就是要求的输出。

上面的分析印证了我们的直觉：在遍历二元查找树时每访问到一个结点，交换它的左右子树。这种思路用递归不难实现，将遍历二元查找树的代码稍作修改就可以了。参考代码如下：

```
//////////////////////////////////////////////////////////////////////
//
// Mirror a BST (swap the left right child of each node) recursively
// the head of BST in initial call
//////////////////////////////////////////////////////////////////////
//
void MirrorRecursively(BSTreeNode *pNode)
{
```

```
    if(!pNode)
        return;

    // swap the right and left child sub-tree
    BSTreeNode *pTemp = pNode->m_pLeft;
    pNode->m_pLeft = pNode->m_pRight;
    pNode->m_pRight = pTemp;

// mirror left child sub-tree if not null
    if(pNode->m_pLeft)
        MirrorRecursively(pNode->m_pLeft);

    // mirror right child sub-tree if not null
    if(pNode->m_pRight)
        MirrorRecursively(pNode->m_pRight);
}
```

由于递归的本质是编译器生成了一个函数调用的栈，因此用循环来完成同样任务时最简单的办法就是用一个辅助栈来模拟递归。首先我们把树的头结点放入栈中。在循环中，只要栈不为空，弹出栈的栈顶结点，交换它的左右子树。如果它有左子树，把它的左子树压入栈中；如果它有右子树，把它的右子树压入栈中。这样在下次循环中就能交换它儿子结点的左右子树了。参考代码如下：

```
//////////////////////////////////////////////////////////////////////
//
// Mirror a BST (swap the left right child of each node) Iteratively
// Input: pTreeHead: the head of BST
//////////////////////////////////////////////////////////////////////
//
void MirrorIteratively(BSTreeNode *pTreeHead)
{
    if(!pTreeHead)
        return;

    std::stack<BSTreeNode*>stackTreeNode;
    stackTreeNode.push(pTreeHead);

while(stackTreeNode.size())
    {
        BSTreeNode *pNode = stackTreeNode.top();
        stackTreeNode.pop();

        // swap the right and left child sub-tree
        BSTreeNode *pTemp = pNode->m_pLeft;
        pNode->m_pLeft = pNode->m_pRight;
        pNode->m_pRight = pTemp;
```

```
        // push left child sub-tree into stack if not null
        if(pNode->m_pLeft)
            stackTreeNode.push(pNode->m_pLeft);

        // push right child sub-tree into stack if not null
        if(pNode->m_pRight)
            stackTreeNode.push(pNode->m_pRight);
    }
}
```

---

(12)一从上往下遍历二元树

题目：输入一颗二元树，从上往下按层打印树的每个结点，同一层中按照从左往右的顺序打印。

例如输入

```
     8
    / \
   6   10
  /\    /\
 5  7  9  11
```

输出 8　6　10　5　7　9　11。

分析：这曾是微软的一道面试题。这道题实质上是要求遍历一棵二元树，只不过不是我们熟悉的前序、中序或者后序遍历。

我们从树的根结点开始分析。自然先应该打印根结点 8，同时为了下次能够打印 8 的两个子结点，我们应该在遍历到 8 时把子结点 6 和 10 保存到一个数据容器中。现在数据容器中就有两个元素 6 和 10 了。按照从左往右的要求，我们先取出 6 访问。打印 6 的同时要把 6 的两个子结点 5 和 7 放入数据容器中，此时数据容器中有三个元素 10、5 和 7。接下来我们应该从数据容器中取出结点 10 访问了。注意 10 比 5 和 7 先放入容器，此时又比 5 和 7 先取出，就是我们通常说的先入先出。因此不难看出这个数据容器的类型应该是个队列。

既然已经确定数据容器是一个队列，现在的问题变成怎么实现队列了。实际上我们无需自己动手实现一个，因为 STL 已经为我们实现了一个很好的 deque（两端都可以进出的队列），我们只需要拿过来用就可以了。

我们知道树是图的一种特殊退化形式。同时如果对图的深度优先遍历和广度优先遍历有比较深刻的理解，将不难看出这种遍历方式实际上是一种广度优先遍历。因此这道题的本质是在二元树上实现广度优先遍历。

参考代码：

```
#include <deque>
#include <iostream>
using namespace std;

struct BTreeNode // a node in the binary tree
{
    int        m_nValue; // value of node
    BTreeNode  *m_pLeft;  // left child of node
    BTreeNode  *m_pRight; // right child of node
};

////////////////////////////////////////////////////////////////////
//
// Print a binary tree from top level to bottom level
// Input: pTreeRoot - the root of binary tree
////////////////////////////////////////////////////////////////////
//
void PrintFromTopToBottom(BTreeNode *pTreeRoot)
{
    if(!pTreeRoot)
        return;

    // get a empty queue
    deque<BTreeNode *> dequeTreeNode;

    // insert the root at the tail of queue
    dequeTreeNode.push_back(pTreeRoot);

    while(dequeTreeNode.size())
    {
        // get a node from the head of queue
        BTreeNode *pNode = dequeTreeNode.front();
        dequeTreeNode.pop_front();

        // print the node
        cout << pNode->m_nValue << ' ';

        // print its left child sub-tree if it has
        if(pNode->m_pLeft)
            dequeTreeNode.push_back(pNode->m_pLeft);
        // print its right child sub-tree if it has
        if(pNode->m_pRight)
```

```
            dequeTreeNode.push_back(pNode->m_pRight);
    }
}
```

PS：层序二叉树，用队列实现！

---

(13)一第一个只出现一次的字符

题目：在一个字符串中找到第一个只出现一次的字符。如输入 abaccdeff，则输出 b。

分析：这道题是 2006 年 google 的一道笔试题。

看到这道题时，最直观的想法是从头开始扫描这个字符串中的每个字符。当访问到某字符时拿这个字符和后面的每个字符相比较，如果在后面没有发现重复的字符，则该字符就是只出现一次的字符。如果字符串有 n 个字符，每个字符可能与后面的 O(n)个字符相比较，因此这种思路时间复杂度是 $O(n^2)$。我们试着去找一个更快的方法。

由于题目与字符出现的次数相关，我们是不是可以统计每个字符在该字符串中出现的次数？要达到这个目的，我们需要一个数据容器来存放每个字符的出现次数。在这个数据容器中可以根据字符来查找它出现的次数，也就是说这个容器的作用是把一个字符映射成一个数字。在常用的数据容器中，哈希表正是这个用途。

哈希表是一种比较复杂的数据结构。由于比较复杂，STL 中没有实现哈希表，因此需要我们自己实现一个。但由于本题的特殊性，我们只需要一个非常简单的哈希表就能满足要求。由于字符（char）是一个长度为 8 的数据类型，因此总共有可能 256 种可能。于是我们创建一个长度为 256 的数组，每个字母根据其 ASCII 码值作为数组的下标对应数组的对应项，而数组中存储的是每个字符对应的次数。这样我们就创建了一个大小为 256，以字符 ASCII 码为键值的哈希表。

我们第一遍扫描这个数组时，每碰到一个字符，在哈希表中找到对应的项并把出现的次数增加一次。这样在进行第二次扫描时，就能直接从哈希表中得到每个字符出现的次数了。

参考代码如下：

```
//////////////////////////////////////////////////////////////////////
//
// Find the first char which appears only once in a string
// Input: pString - the string
// Output: the first not repeating char if the string has, otherwise 0
//////////////////////////////////////////////////////////////////////
//
```

```
char FirstNotRepeatingChar(char* pString)
{
    // invalid input
    if(!pString)
        return 0;

    // get a hash table, and initialize it
        constinttableSize =256;
unsignedinthashTable[tableSize];
for(unsignedinti = 0; i<tableSize; ++ i)
hashTable[i] = 0;

    // get the how many times each char appears in the string
    char* pHashKey = pString;
    while(*(pHashKey) != '\0')
        hashTable[*(pHashKey++)] ++;

    // find the first char which appears only once in a string
    pHashKey = pString;
    while(*pHashKey != '\0')
    {
        if(hashTable[*pHashKey] == 1)
            return *pHashKey;

        pHashKey++;
    }

    // if the string is empty
    // or every char in the string appears at least twice
    return 0;
}
```

---

(14)一圆圈中最后剩下的数字

题目：n 个数字（0,1,...,n-1）形成一个圆圈，从数字 0 开始，每次从这个圆圈中删除第 m 个数字（第一个为当前数字本身，第二个为当前数字的下一个数字）。当一个数字删除后，从被删除数字的下一个继续删除第 m 个数字。求出在这个圆圈中剩下的最后一个数字。

分析：既然题目有一个数字圆圈，很自然的想法是我们用一个数据结构来模拟这个圆圈。在常用的数据结构中，我们很容易想到用环形列表。我们可以创建一个总共有 m 个数字的环形列表，然后每次从这个列表中删除第 m 个元素。

在参考代码中，我们用 STL 中 std::list 来模拟这个环形列表。由于 list 并不是一个环形的结构，因此每次跌代器扫描到列表末尾的时候，要记得把跌代器移到列表的头部。这样就是按照一个圆圈的顺序来遍历这个列表了。

这种思路需要一个有 n 个结点的环形列表来模拟这个删除的过程，因此内存开销为 O(n)。而且这种方法每删除一个数字需要 m 步运算，总共有 n 个数字，因此总的时间复杂度是 O(mn)。当 m 和 n 都很大的时候，这种方法是很慢的。

接下来我们试着从数学上分析出一些规律。首先定义最初的 n 个数字（0,1,…,n-1）中最后剩下的数字是关于 n 和 m 的方程为 f(n,m)。

在这 n 个数字中，第一个被删除的数字是 m%n-1，为简单起见记为 k。那么删除 k 之后的剩下 n-1 的数字为 0,1,…,k-1,k+1,…,n-1，并且下一个开始计数的数字是 k+1。相当于在剩下的序列中，k+1 排到最前面，从而形成序列 k+1,…,n-1,0,…k-1。该序列最后剩下的数字也应该是关于 n 和 m 的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从 0 开始的连续序列），因此该函数不同于前面函数，记为 f'(n-1,m)。最初序列最后剩下的数字 f(n,m)一定是剩下序列的最后剩下数字 f'(n-1,m)，所以 f(n,m)=f'(n-1,m)。

接下来我们把剩下的的这 n-1 个数字的序列 k+1,…,n-1,0,…k-1 作一个映射，映射的结果是形成一个从 0 到 n-2 的序列：

k+1 -> 0
k+2 -> 1
…
n-1 -> n-k-2
0 -> n-k-1
…
k-1 -> n-2

把映射定义为 p，则 p(x)= (x-k-1)%n，即如果映射前的数字是 x，则映射后的数字是(x-k-1)%n。对应的逆映射是 $p^{-1}(x)=(x+k+1)\%n$。

由于映射之后的序列和最初的序列有同样的形式，都是从 0 开始的连续序列，因此仍然可以用函数 f 来表示，记为 f(n-1,m)。根据我们的映射规则，映射之前的序列最后剩下的数字 $f'(n-1,m)= p^{-1}[f(n-1,m)]=[f(n-1,m)+k+1]\%n$。把 k=m%n-1 代入得到 f(n,m)=f'(n-1,m)=[f(n-1,m)+m]%n。

经过上面复杂的分析，我们终于找到一个递归的公式。要得到 n 个数字的序列的最后剩下的数字，只需要得到 n-1 个数字的序列的最后剩下的数字，并可以依此类推。当 n=1 时，也就是序列中开始只有一个数字 0，那么很显然最后剩下的数字就是 0。我们把这种关系表示为：

$$f(n,m)=\begin{cases}0 & n=1\\ [f(n-1,m)+m]\%n & n>1\end{cases}$$

尽管得到这个公式的分析过程非常复杂，但它用递归或者循环都很容易实现。最重要的是，这是一种时间复杂度为 O(n)，空间复杂度为 O(1)的方法，因此无论在时间上还是空间上都优于前面的思路。

思路一的参考代码：

```
////////////////////////////////////////////////////////////////////////
//
// n integers (0, 1, ... n - 1) form a circle. Remove the mth from
// the circle at every time. Find the last number remaining
// Input: n - the number of integers in the circle initially
//        m - remove the mth number at every time
// Output: the last number remaining when the input is valid,
//         otherwise -1
////////////////////////////////////////////////////////////////////////
//
int LastRemaining_Solution1(unsigned int n, unsigned int m)
{
    // invalid input
    if(n < 1 || m < 1)
        return -1;

    unsigned int i = 0;

    // initiate a list with n integers (0, 1, ... n - 1)
    list<int> integers;
    for(i = 0; i < n; ++ i)
        integers.push_back(i);

    list<int>::iterator curinteger = integers.begin();
    while(integers.size() > 1)
    {
        // find the mth integer. Note that std::list is not a circle
        // so we should handle it manually
        for(int i = 1; i < m; ++ i)
        {
            curinteger ++;
            if(curinteger == integers.end())
                curinteger = integers.begin();
        }

        // remove the mth integer. Note that std::list is not a circle
```

```
        // so we should handle it manually
        list<int>::iterator nextinteger = ++ curinteger;
        if(nextinteger == integers.end())
            nextinteger = integers.begin();

        -- curinteger;
        integers.erase(curinteger);
        curinteger = nextinteger;
    }

    return *(curinteger);
}
```

思路二的参考代码：

```
////////////////////////////////////////////////////////////////////////
//
// n integers (0, 1, ... n - 1) form a circle. Remove the mth from
// the circle at every time. Find the last number remaining
// Input: n - the number of integers in the circle initially
//        m - remove the mth number at every time
// Output: the last number remaining when the input is valid,
//         otherwise -1
////////////////////////////////////////////////////////////////////////
//
int LastRemaining_Solution2(int n, unsigned int m)
{
    // invalid input
    if(n <= 0 || m < 0)
        return -1;

    // if there are only one integer in the circle initially,
    // of course the last remaining one is 0
    int lastinteger = 0;

    // find the last remaining one in the circle with n integers
    for (int i = 2; i <= n; i ++)
        lastinteger = (lastinteger + m) % i;

    return lastinteger;
}
```

如果对两种思路的时间复杂度感兴趣的读者可以把 n 和 m 的值设的稍微大一点，比如十万这个数量级的数字，运行的时候就能明显感觉出这两种思路写出来的代码时间效率大不一样。

(15)一含有指针成员的类的拷贝

题目：下面是一个数组类的声明与实现。请分析这个类有什么问题，并针对存在的问题提出几种解决方案。

```
template<typename T> class Array
{
public:
    Array(unsigned arraySize):data(0), size(arraySize)
    {
        if(size > 0)
            data = new T[size];
    }

    ~Array()
    {
        if(data) delete[] data;
    }

    void setValue(unsigned index, const T& value)
    {
        if(index < size)
            data[index] = value;
    }

    T getValue(unsigned index) const
    {
        if(index < size)
            return data[index];
        else
            return T();
    }

private:
    T* data;
    unsigned size;
};
```

分析：我们注意在类的内部封装了用来存储数组数据的指针。软件存在的大部分问题通常都可以归结指针的不正确处理。

这个类只提供了一个构造函数，而没有定义构造拷贝函数和重载拷贝运算符函数。当这个类的用户按照下面的方式声明并实例化该类的一个实例

```
Array A(10);
Array B(A);
```

或者按照下面的方式把该类的一个实例赋值给另外一个实例

```
Array A(10);
Array B(10);
B=A;
```

编译器将调用其自动生成的构造拷贝函数或者拷贝运算符的重载函数。在编译器生成的缺省的构造拷贝函数和拷贝运算符的重载函数，对指针实行的是按位拷贝，仅仅只是拷贝指针的地址，而不会拷贝指针的内容。因此在执行完前面的代码之后，A.data 和 B.data 指向的同一地址。当 A 或者 B 中任意一个结束其生命周期调用析构函数时，会删除 data。由于他们的 data 指向的是同一个地方，两个实例的 data 都被删除了。但另外一个实例并不知道它的 data 已经被删除了，当企图再次用它的 data 的时候，程序就会不可避免地崩溃。

由于问题出现的根源是调用了编译器生成的缺省构造拷贝函数和拷贝运算符的重载函数。一个最简单的办法就是禁止使用这两个函数。于是我们可以把这两个函数声明为私有函数，如果类的用户企图调用这两个函数，将不能通过编译。实现的代码如下：

```
private:
    Array(const Array& copy);
    const Array& operator = (const Array& copy);
```

最初的代码存在问题是因为不同实例的 data 指向的同一地址，删除一个实例的 data 会把另外一个实例的 data 也同时删除。因此我们还可以让构造拷贝函数或者拷贝运算符的重载函数拷贝的不只是地址，而是数据。由于我们重新存储了一份数据，这样一个实例删除的时候，对另外一个实例没有影响。这种思路我们称之为深度拷贝。实现的代码如下：

```
public:
    Array(const Array& copy):data(0), size(copy.size)
    {
        if(size > 0)
        {
            data = new T[size];
            for(int i = 0; i < size; ++ i)
                setValue(i, copy.getValue(i));
        }
    }

    const Array& operator = (const Array& copy)
    {
```

```
        if(this == &copy)
            return *this;

        if(data != NULL)
        {
            delete []data;
            data = NULL;
        }

        size = copy.size;
        if(size > 0)
        {
            data = new T[size];
            for(int i = 0; i < size; ++ i)
                setValue(i, copy.getValue(i));
        }
    }
```

为了防止有多个指针指向的数据被多次删除，我们还可以保存究竟有多少个指针指向该数据。只有当没有任何指针指向该数据的时候才可以被删除。这种思路通常被称之为引用计数技术。在构造函数中，引用计数初始化为 1；每当把这个实例赋值给其他实例或者以参数传给其他实例的构造拷贝函数的时候，引用计数加 1，因为这意味着又多了一个实例指向它的 data；每次需要调用析构函数或者需要把 data 赋值为其他数据的时候，引用计数要减 1，因为这意味着指向它的 data 的指针少了一个。当引用计数减少到 0 的时候，data 已经没有任何实例指向它了，这个时候就可以安全地删除。实现的代码如下：

```
public:
    Array(unsigned arraySize)
        :data(0), size(arraySize), count(new unsigned int)
    {
        *count = 1;
        if(size > 0)
            data = new T[size];
    }

    Array(const Array& copy)
        : size(copy.size), data(copy.data), count(copy.count)
    {
        ++ (*count);
    }

    ~Array()
    {
        Release();
    }
```

```
    const Array& operator = (const Array& copy)
    {
        if(data == copy.data)
            return *this;

        Release();

        data = copy.data;
        size = copy.size;
        count = copy.count;
        ++(*count);
    }

private:
    void Release()
    {
        --(*count);
        if(*count == 0)
        {
            if(data)
            {
                delete []data;
                data = NULL;
            }

            delete count;
            count = 0;
        }
    }

    unsigned int *count;
```

PS：拷贝构造函数和重载拷贝运算符函数，深复制，避免调用缺省（浅复制），两对象指向同一地址，删除一个时出现错误！

---

(16)－O(logn)求 Fibonacci 数列

题目：定义 Fibonacci 数列如下：

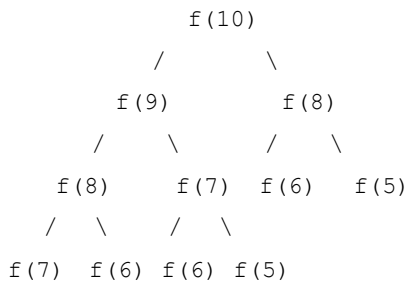
```
       /  0                    n=0
f(n)=      1                    n=1
       \  f(n-1)+f(n-2)          n=2
```

输入 n，用最快的方法求该数列的第 n 项。

分析：在很多 C 语言教科书中讲到递归函数的时候，都会用 Fibonacci 作为例子。因此很多程序员对这道题的递归解法非常熟悉，看到题目就能写出如下的递归求解的代码。

```
////////////////////////////////////////////////////////////////////////
//
// Calculate the nth item of Fibonacci Series recursively
////////////////////////////////////////////////////////////////////////
//
long long Fibonacci_Solution1(unsigned int n)
{
    int result[2] = {0, 1};
    if(n < 2)
        return result[n];

    return Fibonacci_Solution1(n - 1) + Fibonacci_Solution1(n - 2);
}
```

但是，教科书上反复用这个题目来讲解递归函数，并不能说明递归解法最适合这道题目。我们以求解 f(10)作为例子来分析递归求解的过程。要求得 f(10)，需要求得 f(9)和 f(8)。同样，要求得 f(9)，要先求得 f(8)和 f(7)……我们用树形结构来表示这种依赖关系



我们不难发现在这棵树中有很多结点会重复的，而且重复的结点数会随着 n 的增大而急剧增加。这意味这计算量会随着 n 的增大而急剧增大。事实上，用递归方法计算的时间复杂度是以 n 的指数的方式递增的。大家可以求 Fibonacci 的第 100 项试试，感受一下这样递归会慢到什么程度。在我的机器上，连续运行了一个多小时也没有出来结果。

其实改进的方法并不复杂。上述方法之所以慢是因为重复的计算太多，只要避免重复计算就行了。比如我们可以把已经得到的数列中间项保存起来，如果下次需要计算的时候我们先查找一下，如果前面已经计算过了就不用再次计算了。

更简单的办法是从下往上计算，首先根据 f(0)和 f(1)算出 f(2)，在根据 f(1)和 f(2)算出 f(3)……依此类推就可以算出第 n 项了。很容易理解，这种思路的时间复杂度是 O(n)。

```
//////////////////////////////////////////////////////////////////////
//
// Calculate the nth item of Fibonacci Series iteratively
//////////////////////////////////////////////////////////////////////
//
long long Fibonacci_Solution2(unsigned n)
{
    int result[2] = {0, 1};
    if(n < 2)
        return result[n];

    long long  fibNMinusOne = 1;
    long long  fibNMinusTwo = 0;
    long long  fibN = 0;
    for(unsigned int i = 2; i <= n; ++ i)
    {
        fibN = fibNMinusOne + fibNMinusTwo;

        fibNMinusTwo = fibNMinusOne;
        fibNMinusOne = fibN;
    }

    return fibN;
}
```

这还不是最快的方法。下面介绍一种时间复杂度是 O(logn)的方法。在介绍这种方法之前，先介绍一个数学公式：

$\{f(n), f(n-1), f(n-1), f(n-2)\} = \{1, 1, 1, 0\}^{n-1}$

(注：{f(n+1), f(n), f(n), f(n-1)}表示一个矩阵。在矩阵中第一行第一列是 f(n+1)，第一行第二列是 f(n)，第二行第一列是 f(n)，第二行第二列是 f(n-1)。)

有了这个公式，要求得 f(n)，我们只需要求得矩阵{1, 1, 1,0}的 n-1 次方，因为矩阵{1, 1, 1,0}的 n-1 次方的结果的第一行第一列就是 f(n)。这个数学公式用数学归纳法不难证明。感兴趣的朋友不妨自己证明一下。

现在的问题转换为求矩阵{1, 1, 1, 0}的乘方。如果简单第从 0 开始循环，n 次方将需要 n 次运算，并不比前面的方法要快。但我们可以考虑乘方的如下性质：

$$a^n = \begin{cases} a^{n/2} * a^{n/2} & n \text{ 为偶数时} \\ a^{(n-1)/2} * a^{(n-1)/2} & n \text{ 为奇数时} \end{cases}$$

要求得 n 次方，我们先求得 n/2 次方，再把 n/2 的结果平方一下。如果把求 n 次方的问题看成一个大问题，把求 n/2 看成一个较小的问题。这种把大问题分解成一个或多个小问题的思路我们称之为分治法。这样求 n 次方就只需要 logn 次运算了。

实现这种方式时，首先需要定义一个 2×2 的矩阵，并且定义好矩阵的乘法以及乘方运算。当这些运算定义好了之后，剩下的事情就变得非常简单。完整的实现代码如下所示。

```
#include <cassert>

//////////////////////////////////////////////////////////////////////
//
// A 2 by 2 matrix
//////////////////////////////////////////////////////////////////////
//
struct Matrix2By2
{
    Matrix2By2
    (
        long long m00 = 0,
        long long m01 = 0,
        long long m10 = 0,
        long long m11 = 0
    )
    :m_00(m00), m_01(m01), m_10(m10), m_11(m11)
    {
    }

    long long m_00;
    long long m_01;
    long long m_10;
    long long m_11;
};

//////////////////////////////////////////////////////////////////////
//
// Multiply two matrices
// Input: matrix1 - the first matrix
//        matrix2 - the second matrix
//Output: the production of two matrices
//////////////////////////////////////////////////////////////////////
//
Matrix2By2 MatrixMultiply
(
    const Matrix2By2& matrix1,
    const Matrix2By2& matrix2
```

```
)
{
    return Matrix2By2(
        matrix1.m_00 * matrix2.m_00 + matrix1.m_01 * matrix2.m_10,
        matrix1.m_00 * matrix2.m_01 + matrix1.m_01 * matrix2.m_11,
        matrix1.m_10 * matrix2.m_00 + matrix1.m_11 * matrix2.m_10,
        matrix1.m_10 * matrix2.m_01 + matrix1.m_11 * matrix2.m_11);
}

////////////////////////////////////////////////////////////////////
//
// The nth power of matrix
// 1  1
// 1  0
////////////////////////////////////////////////////////////////////
//
Matrix2By2 MatrixPower(unsigned int n)
{
    assert(n > 0);

    Matrix2By2 matrix;
    if(n == 1)
    {
        matrix = Matrix2By2(1, 1, 1, 0);
    }
    else if(n % 2 == 0)
    {
        matrix = MatrixPower(n / 2);
        matrix = MatrixMultiply(matrix, matrix);
    }
    else if(n % 2 == 1)
    {
        matrix = MatrixPower((n - 1) / 2);
        matrix = MatrixMultiply(matrix, matrix);
        matrix = MatrixMultiply(matrix, Matrix2By2(1, 1, 1, 0));
    }

    return matrix;
}

////////////////////////////////////////////////////////////////////
//
// Calculate the nth item of Fibonacci Series using devide and conquer
////////////////////////////////////////////////////////////////////
```

```
//
long long Fibonacci_Solution3(unsigned int n)
{
    int result[2] = {0, 1};
    if(n < 2)
        return result[n];

    Matrix2By2 PowerNMinus2 = MatrixPower(n - 1);
    return PowerNMinus2.m_00;
}
```

---

(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;
        longlongnum = 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。

---

(18)—用两个栈实现队列

题目：某队列的声明如下：

```
template<typename T> class CQueue
{
public:
    CQueue() {}
    ~CQueue() {}

    void appendTail(const T& node);  // append a element to tail
    void deleteHead();               // remove a element from head

private:
    T>m_stack1;
    T>m_stack2;
};
```

分析：从上面的类的声明中，我们发现在队列中有两个栈。因此这道题实质上是要求我们用两个栈来实现一个队列。相信大家对栈和队列的基本性质都非常了解了：栈是一种后入先出的数据容器，因此对队列进行的插入和删除操作都是在栈顶上进行；队列是一种先入先出的数据容器，我们总是把新元素插入到队列的尾部，而从队列的头部删除元素。

我们通过一个具体的例子来分析往该队列插入和删除元素的过程。首先插入一个元素 a，不妨把先它插入到 m_stack1。这个时候 m_stack1 中的元素有{a}，m_stack2 为空。再插入两个元素 b 和 c，还是插入到 m_stack1 中，此时 m_stack1 中的元素有{a,b,c}，m_stack2 中仍然是空的。

这个时候我们试着从队列中删除一个元素。按照队列先入先出的规则，由于 a 比 b、c 先插入到队列中，这次被删除的元素应该是 a。元素 a 存储在 m_stack1 中，但并不在栈顶上，因此不能直接进行删除。注意到 m_stack2 我们还一直没有使用过，现在是让 m_stack2 起作用的时候了。如果我们把 m_stack1 中的元素逐个 pop 出来并 push 进入 m_stack2，元素在 m_stack2 中的顺序正好和原来在 m_stack1 中的顺序相反。因此经过两次 pop 和 push 之后，m_stack1 为空，而 m_stack2 中的元素是{c,b,a}。这个时候就可以 pop 出 m_stack2 的栈顶 a 了。pop 之后的 m_stack1 为空，而 m_stack2 的元素为{c,b}，其中 b 在栈顶。

这个时候如果我们还想继续删除应该怎么办呢？在剩下的两个元素中 b 和 c，b 比 c 先进入队列，因此 b 应该先删除。而此时 b 恰好又在栈顶上，因此可以直接 pop 出去。这次 pop 之后，m_stack1 中仍然为空，而 m_stack2 为{c}。

从上面的分析我们可以总结出删除一个元素的步骤：当 m_stack2 中不为空时，在 m_stack2 中的栈顶元素是最先进入队列的元素，可以 pop 出去。如果 m_stack2 为空时，我们把 m_stack1 中的元素逐个 pop 出来并 push 进入 m_stack2。由于先进入队列的元素被压到 m_stack1 的底端，经过 pop 和 push 之后就处于 m_stack2 的顶端了，又可以直接 pop 出去。

接下来我们再插入一个元素 d。我们是不是还可以把它 push 进 m_stack1？这样会不会有问题呢？我们说不会有问题。因为在删除元素的时候，如果 m_stack2 中不为空，处于 m_stack2 中的栈顶元素是最先进入队列的，可以直接 pop；如果 m_stack2 为空，我们把 m_stack1 中的元素 pop 出来并 push 进入 m_stack2。由于 m_stack2 中元素的顺序和 m_stack1 相反，最先进入队列的元素还是处于 m_stack2 的栈顶，仍然可以直接 pop。不会出现任何矛盾。

我们用一个表来总结一下前面的例子执行的步骤：

操作	m_stack1	m_stack2
append a	{a}	{}
append b	{a,b}	{}
append c	{a,b,c}	{}
delete head	{}	{b,c}
delete head	{}	{c}
append d	{d}	{c}
delete head	{d}	{}

总结完 push 和 pop 对应的过程之后，我们可以开始动手写代码了。参考代码如下：

```
//////////////////////////////////////////////////////////////////////
//
// Append a element at the tail of the queue
//////////////////////////////////////////////////////////////////////
//
template<typename T> void CQueue<T>::appendTail(const T& element)
{
    // push the new element into m_stack1
    m_stack1.push(element);
}

//////////////////////////////////////////////////////////////////////
//
// Delete the head from the queue
//////////////////////////////////////////////////////////////////////
//
```

```
template<typename T> void CQueue<T>::deleteHead()
{
    // if m_stack2is empty,and there are some
//elements inm_stack1, push them in m_stack2
if(m_stack2.size()<= 0)
{
while(m_stack1.size()>0)
{
T&data =m_stack1.top();
m_stack1.pop();
m_stack2.push(data);
}
}

// push theelement into m_stack2
assert(m_stack2.size()>0);
m_stack2.pop();
}
```

扩展：这道题是用两个栈实现一个队列。反过来能不能用两个队列实现一个栈？如果可以，该如何实现？

---

(19)—反转链表

题目：输入一个链表的头结点，反转该链表，并返回反转后链表的头结点。链表结点定义如下：

```
struct ListNode
{
    int      m_nKey;
    ListNode* m_pNext;
};
```

分析：这是一道广为流传的微软面试题。由于这道题能够很好的反应出程序员思维是否严密，在微软之后已经有很多公司在面试时采用了这道题。

为了正确地反转一个链表，需要调整指针的指向。与指针操作相关代码总是容易出错的，因此最好在动手写程序之前作全面的分析。在面试的时候不急于动手而是一开始做仔细的分析和设计，将会给面试官留下很好的印象，因为在实际的软件开发中，设计的时间总是比写代码的时间长。与其很快地写出一段漏洞百出的代码，远不如用较多的时间写出一段健壮的代码。

为了将调整指针这个复杂的过程分析清楚，我们可以借助图形来直观地分析。假设下图中 l、m 和 n 是三个相邻的结点：

a←b←...←l m→n→...

假设经过若干操作，我们已经把结点 l 之前的指针调整完毕，这些结点的 m_pNext 指针都指向前面一个结点。现在我们遍历到结点 m。当然，我们需要把调整结点的 m_pNext 指针让它指向结点 l。但注意一旦调整了指针的指向，链表就断开了，如下图所示：

a←b←...l←m n→...

因为已经没有指针指向结点 n，我们没有办法再遍历到结点 n 了。因此为了避免链表断开，我们需要在调整 m 的 m_pNext 之前要把 n 保存下来。

接下来我们试着找到反转后链表的头结点。不难分析出反转后链表的头结点是原始链表的尾位结点。什么结点是尾结点？就是 m_pNext 为空指针的结点。

基于上述分析，我们不难写出如下代码：

```
////////////////////////////////////////////////////////////////////
//
// Reverse a list iteratively
// Input: pHead - the head of the original list
// Output: the head of the reversed head
////////////////////////////////////////////////////////////////////
//
ListNode* ReverseIteratively(ListNode* pHead)
{
    ListNode* pReversedHead = NULL;
    ListNode* pNode = pHead;
    ListNode* pPrev = NULL;
    while(pNode != NULL)
    {
        // get the next node, and save it at pNext
        ListNode* pNext = pNode->m_pNext;

        // if the next node is null, the currect is the end of original
        // list, and it's the head of the reversed list
        if(pNext == NULL)
            pReversedHead = pNode;

        // reverse the linkage between nodes
        pNode->m_pNext = pPrev;

        // move forward on the the list
```

```
        pPrev = pNode;
        pNode = pNext;
    }

    return pReversedHead;
}
```

扩展：本题也可以递归实现。感兴趣的读者请自己编写递归代码。

---

(20)一最长公共子串

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 $X_m=\{x_0, x_1,\ldots x_{m-1}\}$和 $Y_n=\{y_0,y_1,\ldots,y_{n-1}\}$为两个字符串，而 $Z_k=\{z_0,z_1,\ldots z_{k-1}\}$是它们的 LCS，则：

1. 如果 $x_{m-1}=y_{n-1}$，那么 $z_{k-1}=x_{m-1}=y_{n-1}$，并且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的 LCS；
2. 如果 $x_{m-1}\neq y_{n-1}$，那么当 $z_{k-1}\neq x_{m-1}$ 时 Z 是 X_{m-1} 和 Y 的 LCS；
3. 如果 $x_{m-1}\neq y_{n-1}$，那么当 $z_{k-1}\neq y_{n-1}$ 时 Z 是 Y_{n-1} 和 X 的 LCS；

下面简单证明一下这些性质：

1. 如果 $z_{k-1}\neq x_{m-1}$，那么我们可以把 x_{m-1}（y_{n-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 $z_{k-1}=x_{m-1}=y_{n-1}$。

既然 $z_{k-1}=x_{m-1}=y_{n-1}$，那如果我们删除 z_{k-1}（x_{m-1}、y_{n-1}）得到的 Z_{k-1}，X_{m-1} 和 Y_{n-1}，显然 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个公共子串，现在我们证明 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的 LCS。用反证法不难证明。假设有 X_{m-1} 和 Y_{n-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 X_{m-1} 和 Y 的 LCS，则存在一个长度超过 k 的 W 是 X_{m-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 $X_m=\{x_0, x_1,\ldots x_{m-1}\}$和 $Y_n=\{y_0,y_1,\ldots,y_{n-1}\}$的 LCS，如果 $x_{m-1}=y_{n-1}$，那么只需求得 X_{m-1} 和 Y_{n-1} 的 LCS，并在其后添加 x_{m-1}（y_{n-1}）即可；如果 $x_{m-1}\neq y_{n-1}$，我们分别求得 X_{m-1} 和 Y 的 LCS 和 Y_{n-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 X_i 和 Y_j 的 LCS 的长度为 c[i,j]，我们可以递归地求 c[i,j]：

$$c[i,j]=\begin{cases}0 & \text{if } i<0 \text{ or } j<0\\ c[i-1,j-1]+1 & \text{if } i,j>=0 \text{ and } x_i=x_j\\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>=0 \text{ and } x_i\neq x_j\end{cases}$$

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[i] = (int*)new int[length2];

    for(i = 0; i < length1; ++ i)
        for(j = 0; j < length2; ++ j)
            LCS_length[i][j] = 0;



    // initiate the direction matrix
    int **LCS_direction;
    LCS_direction = (int**)(new int[length1]);
    for( i = 0; i < length1; ++ i)
        LCS_direction[i] = (int*)new int[length2];

    for(i = 0; i < length1; ++ i)
        for(j = 0; j < length2; ++ j)
            LCS_direction[i][j] = kInit;

    for(i = 0; i < length1; ++ i)
    {
        for(j = 0; j < length2; ++ j)
        {
            if(i == 0 || j == 0)
            {
                if(pStr1[i] == pStr2[j])
                {
                    LCS_length[i][j] = 1;
                    LCS_direction[i][j] = kLeftUp;
                }
                else
                    LCS_length[i][j] = 0;
            }
            // a char of LCS is found,
            // it comes from the left up entry in the direction matrix
```

```
            else if(pStr1[i] == pStr2[j])
            {
                LCS_length[i][j] = LCS_length[i - 1][j - 1] + 1;
                LCS_direction[i][j] = kLeftUp;
            }
            // it comes from the up entry in the direction matrix
            else if(LCS_length[i - 1][j] > LCS_length[i][j - 1])
            {
                LCS_length[i][j] = LCS_length[i - 1][j];
                LCS_direction[i][j] = kUp;
            }
            // it comes from the left entry in the direction matrix
            else
            {
                LCS_length[i][j] = LCS_length[i][j - 1];
                LCS_direction[i][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。

---

(21)一左旋转字符串

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 X^T。显然有$(X^T)^T=X$。

我们首先对 X 和 Y 两段分别进行翻转操作，这样就能得到 X^TY^T。接着再对 X^TY^T 进行翻转操作，得到$(X^TY^T)^T=(Y^T)^T(X^T)^T=YX$。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 --;
        }
    }
}
```

PS：$(X^TY^T)^T=(Y^T)^T(X^T)^T=YX$

---

(22)一整数的二进制表示中 1 的个数

题目：输入一个整数，求该整数的二进制表达中有多少个 1。例如输入 10，由于其二进制表示为 1010，有两个 1，因此输出 2。

分析：这是一道很基本的考查位运算的面试题。包括微软在内的很多公司都曾采用过这道题。

一个很基本的想法是，我们先判断整数的最右边一位是不是 1。接着把整数右移一位，原来处于右边第二位的数字现在被移到第一位了，再判断是不是 1。这样每次移动一位，直到这个整数变成 0 为止。现在的问题变成怎样判断一个整数的最右边一位是不是 1 了。很简单，如果它和整数 1 作与运算。由于 1 除了最右边一位以外，其他所有位都为 0。因此如果与运算的结果为 1，表示整数的最右边一位是 1，否则是 0。

得到的代码如下：

```
////////////////////////////////////////////////////////////////////////
//
// Get how many 1s in an integer's binary expression
////////////////////////////////////////////////////////////////////////
//
int NumberOf1_Solution1(int i)
{
    int count = 0;
    while(i)
    {
        if(i & 1)
            count ++;

        i = i >> 1;
    }

    return count;
}
```

可能有读者会问，整数右移一位在数学上是和除以 2 是等价的。那可不可以把上面的代码中的右移运算符换成除以 2 呢？答案是最好不要换成除法。因为除法的效率比移位运算要低的多，在实际编程中如果可以应尽可能地用移位运算符代替乘除法。

这个思路当输入 i 是正数时没有问题，但当输入的 i 是一个负数时，不但不能得到正确的 1 的个数，还将导致死循环。以负数 0x80000000 为例，右移一位的时候，并不是简单地把最高位的 1 移到第二位变成 0x40000000，而是 0xC0000000。这是因为移位前是个负数，仍然要保证移位后是个负数，因此移位后的最高位会设为 1。如果一直做右移运算，最终这个数字就会变成 0xFFFFFFFF 而陷入死循环。

为了避免死循环，我们可以不右移输入的数字 i。首先 i 和 1 做与运算，判断 i 的最低位是不是为 1。接着把 1 左移一位得到 2，再和 i 做与运算，就能判断 i 的次高位是不是 1……这样反复左移，每次都能判断 i 的其中一位是不是 1。基于此，我们得到如下代码：

```
////////////////////////////////////////////////////////////////////////
//
// Get how many 1s in an integer's binary expression
```

```
////////////////////////////////////////////////////////////////////////
//
int NumberOf1_Solution2(int i)
{
    int count = 0;
    unsigned int flag = 1;
    while(flag)
    {
        if(i & flag)
            count ++;

        flag = flag << 1;
    }

    return count;
}
```

另外一种思路是如果一个整数不为 0，那么这个整数至少有一位是 1。如果我们把这个整数减去 1，那么原来处在整数最右边的 1 就会变成 0，原来在 1 后面的所有的 0 都会变成 1。其余的所有位将不受到影响。举个例子：一个二进制数 1100，从右边数起的第三位是处于最右边的一个 1。减去 1 后，第三位变成 0，它后面的两位 0 变成 1，而前面的 1 保持不变，因此得到结果是 1011。

我们发现减 1 的结果是把从最右边一个 1 开始的所有位都取反了。这个时候如果我们再把原来的整数和减去 1 之后的结果做与运算，从原来整数最右边一个 1 那一位开始所有位都会变成 0。如 1100&1011=1000。也就是说，把一个整数减去 1，再和原整数做与运算，会把该整数最右边一个 1 变成 0。那么一个整数的二进制有多少个 1，就可以进行多少次这样的操作。

这种思路对应的代码如下：

```
////////////////////////////////////////////////////////////////////////
//
// Get how many 1s in an integer's binary expression
////////////////////////////////////////////////////////////////////////
//
int NumberOf1_Solution3(int i)
{
    int count = 0;

    while (i)
    {
        ++ count;
        i = (i - 1) & i;
    }
```

```
    return count;
}
```

扩展：如何用一个语句判断一个整数是不是二的整数次幂？

PS：n&(n-1)==0;//二进制数只有一位位 1，则该数是 2 的整数次幂.

---

(23)一跳台阶问题

题目：一个台阶总共有 n 级，如果一次可以跳 1 级，也可以跳 2 级。求总共有多少总跳法，并分析算法的时间复杂度。

分析：这道题最近经常出现，包括 MicroStrategy 等比较重视算法的公司都曾先后选用过个这道题作为面试题或者笔试题。

首先我们考虑最简单的情况。如果只有 1 级台阶，那显然只有一种跳法。如果有 2 级台阶，那就有两种跳的方法了：一种是分两次跳，每次跳 1 级；另外一种就是一次跳 2 级。

现在我们再来讨论一般情况。我们把 n 级台阶时的跳法看成是 n 的函数，记为 f(n)。当 n>2 时，第一次跳的时候就有两种不同的选择：一是第一次只跳 1 级，此时跳法数目等于后面剩下的 n-1 级台阶的跳法数目，即为 f(n-1)；另外一种选择是第一次跳 2 级，此时跳法数目等于后面剩下的 n-2 级台阶的跳法数目，即为 f(n-2)。因此 n 级台阶时的不同跳法的总数 f(n)=f(n-1)+(f-2)。

我们把上面的分析用一个公式总结如下：

```
        /  1                      n=1
f(n)=       2                      n=2
        \  f(n-1)+(f-2)              n>2
```

分析到这里，相信很多人都能看出这就是我们熟悉的 Fibonacci 序列。至于怎么求这个序列的第 n 项，请参考本面试题系列第 16 题，这里就不在赘述了。

---

(24)一栈的 push、pop 序列

题目：输入两个整数序列。其中一个序列表示栈的 push 顺序，判断另一个序列有没有可能是对应的 pop 顺序。为了简单起见，我们假设 push 序列的任意两个整数都是不相等的。

比如输入的 push 序列是 1、2、3、4、5，那么 4、5、3、2、1 就有可能是一个 pop 系列。因为可以有如下的 push 和 pop 序列：push 1，push 2，push 3，push 4，pop，push 5，pop，pop，pop，pop，这样得到的 pop 序列就是 4、5、3、2、1。但序列 4、3、5、1、2 就不可能是 push 序列 1、2、3、4、5 的 pop 序列。

分析：这到题除了考查对栈这一基本数据结构的理解，还能考查我们的分析能力。

这道题的一个很直观的想法就是建立一个辅助栈，每次 push 的时候就把一个整数 push 进入这个辅助栈，同样需要 pop 的时候就把该栈的栈顶整数 pop 出来。

我们以前面的序列 4、5、3、2、1 为例。第一个希望被 pop 出来的数字是 4，因此 4 需要先 push 到栈里面。由于 push 的顺序已经由 push 序列确定了，也就是在把 4 push 进栈之前，数字 1，2，3 都需要 push 到栈里面。此时栈里的包含 4 个数字，分别是 1，2，3，4，其中 4 位于栈顶。把 4 pop 出栈后，剩下三个数字 1，2，3。接下来希望被 pop 的是 5，由于仍然不是栈顶数字，我们接着在 push 序列中 4 以后的数字中寻找。找到数字 5 后再一次 push 进栈，这个时候 5 就是位于栈顶，可以被 pop 出来。接下来希望被 pop 的三个数字是 3，2，1。每次操作前都位于栈顶，直接 pop 即可。

再来看序列 4、3、5、1、2。pop 数字 4 的情况和前面一样。把 4 pop 出来之后，3 位于栈顶，直接 pop。接下来希望 pop 的数字是 5，由于 5 不是栈顶数字，我们到 push 序列中没有被 push 进栈的数字中去搜索该数字，幸运的时候能够找到 5，于是把 5 push 进入栈。此时 pop 5 之后，栈内包含两个数字 1、2，其中 2 位于栈顶。这个时候希望 pop 的数字是 1，由于不是栈顶数字，我们需要到 push 序列中还没有被 push 进栈的数字中去搜索该数字。但此时 push 序列中所有数字都已被 push 进入栈，因此该序列不可能是一个 pop 序列。

也就是说，如果我们希望 pop 的数字正好是栈顶数字，直接 pop 出栈即可；如果希望 pop 的数字目前不在栈顶，我们就到 push 序列中还没有被 push 到栈里的数字中去搜索这个数字，并把在它之前的所有数字都 push 进栈。如果所有的数字都被 push 进栈仍然没有找到这个数字，表明该序列不可能是一个 pop 序列。

基于前面的分析，我们可以写出如下的参考代码：

```
#include <stack>

///////////////////////////////////////////////////////////////////////
////////
// Given a push order of a stack, determine whether an array is possible
to
// be its corresponding pop order
// Input: pPush   - an array of integers, the push order
//        pPop    - an array of integers, the pop order
//        nLength - the length of pPush and pPop
```

```
// Output: If pPop is possible to be the pop order of pPush, return true.
//        Otherwise return false
////////////////////////////////////////////////////////////////////////
////////
bool IsPossiblePopOrder(const int* pPush, const int* pPop, int nLength)
{
    bool bPossible = false;

    if(pPush && pPop && nLength > 0)
    {
        const int *pNextPush = pPush;
        const int *pNextPop = pPop;

        // ancillary stack
        std::stack<int>stackData;

        // check every integers in pPop
        while(pNextPop - pPop < nLength)
        {
            // while the top of the ancillary stack is not the integer
            // to be poped, try to push some integers into the stack
            while(stackData.empty() || stackData.top() != *pNextPop)
            {
                // pNextPush == NULL means all integers have been
                // pushed into the stack, can't push any longer
                if(!pNextPush)
                    break;

                stackData.push(*pNextPush);

                // if there are integers left in pPush, move
                // pNextPush forward, otherwise set it to be NULL
                if(pNextPush - pPush < nLength - 1)
                    pNextPush ++;
                else
                    pNextPush = NULL;
            }

            // After pushing, the top of stack is still not same as
            // pPextPop, pPextPop is not in a pop sequence
            // corresponding to pPush
            if(stackData.top() != *pNextPop)
                break;
```

```
            // Check the next integer in pPop
            stackData.pop();
            pNextPop ++;
        }

        // if all integers in pPop have been check successfully,
        // pPop is a pop sequence corresponding to pPush
        if(stackData.empty() && pNextPop - pPop == nLength)
            bPossible = true;
    }

    return bPossible;
}
```

---

(25)-在从 1 到 n 的正数中 1 出现的次数

题目：输入一个整数 n，求从 1 到 n 这 n 个整数的十进制表示中 1 出现的次数。

例如输入 12，从 1 到 12 这些整数中包含 1 的数字有 1，10，11 和 12，1 一共出现了 5 次。

分析：这是一道广为流传的 google 面试题。用最直观的方法求解并不是很难，但遗憾的是效率不是很高；而要得出一个效率较高的算法，需要比较强的分析能力，并不是件很容易的事情。当然，google 的面试题中简单的也没有几道。

首先我们来看最直观的方法，分别求得 1 到 n 中每个整数中 1 出现的次数。而求一个整数的十进制表示中 1 出现的次数，就和本面试题系列的第 22 题很相像了。我们每次判断整数的个位数字是不是 1。如果这个数字大于 10，除以 10 之后再判断个位数字是不是 1。基于这个思路，不难写出如下的代码：

```
int NumberOf1(unsigned int n);

////////////////////////////////////////////////////////////////////////
////////
// Find the number of 1 in the integers between 1 and n
// Input: n - an integer
// Output: the number of 1 in the integers between 1 and n
////////////////////////////////////////////////////////////////////////
////////
int NumberOf1BeforeBetween1AndN_Solution1(unsigned int n)
{
    int number = 0;
```

```
    // Find the number of 1 in each integer between 1 and n
    for(unsigned int i = 1; i <= n; ++ i)
        number += NumberOf1(i);

    return number;
}

////////////////////////////////////////////////////////////////////////
////////
// Find the number of 1 in an integer with radix 10
// Input: n - an integer
// Output: the number of 1 in n with radix
////////////////////////////////////////////////////////////////////////
////////
int NumberOf1(unsigned int n)
{
    int number = 0;
    while(n)
    {
        if(n % 10 == 1)
            number ++;

        n = n / 10;
    }

    return number;
}
```

这个思路有一个非常明显的缺点就是每个数字都要计算 1 在该数字中出现的次数，因此时间复杂度是 O(n)。当输入的 n 非常大的时候，需要大量的计算，运算效率很低。我们试着找出一些规律，来避免不必要的计算。

我们用一个稍微大一点的数字 21345 作为例子来分析。我们把从 1 到 21345 的所有数字分成两段，即 1-1235 和 1346-21345。

先来看 1346-21345 中 1 出现的次数。1 的出现分为两种情况：一种情况是 1 出现在最高位（万位）。从 1 到 21345 的数字中，1 出现在 10000-19999 这 10000 个数字的万位中，一共出现了 10000（10^4）次；另外一种情况是 1 出现在除了最高位之外的其他位中。例子中 1346-21345，这 20000 个数字中后面四位中 1 出现的次数是 2000 次（$2*10^3$，其中 2 的第一位的数值，10^3 是因为数字的后四位数字其中一位为 1，其余的三位数字可以在 0 到 9 这 10 个数字任意选择，由排列组合可以得出总次数是 $2*10^3$）。

至于从 1 到 1345 的所有数字中 1 出现的次数，我们就可以用递归地求得了。这也是我们为什么要把 1-21345 分为 1-1235 和 1346-21345 两段的原因。因为把 21345 的最高位去掉就得到 1345，便于我们采用递归的思路。

分析到这里还有一种特殊情况需要注意：前面我们举例子是最高位是一个比 1 大的数字，此时最高位 1 出现的次数 10^4（对五位数而言）。但如果最高位是 1 呢？比如输入 12345，从 10000 到 12345 这些数字中，1 在万位出现的次数就不是 10^4 次，而是 2346 次了，也就是除去最高位数字之后剩下的数字再加上 1。

基于前面的分析，我们可以写出以下的代码。在参考代码中，为了编程方便，我把数字转换成字符串了。

```
#include "string.h"
#include "stdlib.h"

int NumberOf1(const char* strN);
int PowerBase10(unsigned int n);

////////////////////////////////////////////////////////////////////////
////////
// Find the number of 1 in an integer with radix 10
// Input: n - an integer
// Output: the number of 1 in n with radix
////////////////////////////////////////////////////////////////////////
////////
int NumberOf1BeforeBetween1AndN_Solution2(int n)
{
    if(n <= 0)
        return 0;

    // convert the integer into a string
    char strN[50];
    sprintf(strN, "%d", n);

    return NumberOf1(strN);
}

////////////////////////////////////////////////////////////////////////
////////
// Find the number of 1 in an integer with radix 10
// Input: strN - a string, which represents an integer
// Output: the number of 1 in n with radix
////////////////////////////////////////////////////////////////////////
////////
int NumberOf1(const char* strN)
{
    if(!strN || *strN < '0' || *strN > '9' || *strN == '\0')
        return 0;

    int firstDigit = *strN - '0';
    unsigned int length = static_cast<unsigned int>(strlen(strN));
```

```
    // the integer contains only one digit
    if(length == 1 && firstDigit == 0)
        return 0;

    if(length == 1 && firstDigit > 0)
        return 1;

    // suppose the integer is 21345
    // numFirstDigit is the number of 1 of 10000-19999 due to the first
digit
    int numFirstDigit = 0;
    // numOtherDigits is the number of 1 01346-21345 due to all digits
    // except the first one
    int numOtherDigits = firstDigit * (length - 1) * PowerBase10(length
- 2);
    // numRecursive is the number of 1 of integer 1345
    int numRecursive = NumberOf1(strN + 1);

    // if the first digit is greater than 1, suppose in integer 21345
    // number of 1 due to the first digit is 10^4. It's 10000-19999
    if(firstDigit > 1)
        numFirstDigit = PowerBase10(length - 1);

    // if the first digit equals to 1, suppose in integer 12345
    // number of 1 due to the first digit is 2346. It's 10000-12345
    else if(firstDigit == 1)
        numFirstDigit = atoi(strN + 1) + 1;

    return numFirstDigit + numOtherDigits + numRecursive;
}

////////////////////////////////////////////////////////////////////////
////////
// Calculate 10^n
////////////////////////////////////////////////////////////////////////
////////
int PowerBase10(unsigned int n)
{
    int result = 1;
    for(unsigned int i = 0; i < n; ++ i)
        result *= 10;
```

```
    return result;
}
```

---

(26)-和为 n 连续正数序列

题目：输入一个正数 n，输出所有和为 n 连续正数序列。

例如输入 15，由于 1+2+3+4+5=4+5+6=7+8=15，所以输出 3 个连续序列 1-5、4-6 和 7-8。

分析：这是网易的一道面试题。

这道题和本面试题系列的第 10 题有些类似。我们用两个数 small 和 big 分别表示序列的最小值和最大值。首先把 small 初始化为 1，big 初始化为 2。如果从 small 到 big 的序列的和大于 n 的话，我们向右移动 small，相当于从序列中去掉较小的数字。如果从 small 到 big 的序列的和小于 n 的话，我们向右移动 big，相当于向序列中添加 big 的下一个数字。一直到 small 等于(1+n)/2，因为序列至少要有两个数字。

基于这个思路，我们可以写出如下代码：

```
void PrintContinuousSequence(int small, int big);

//////////////////////////////////////////////////////////////////////
////
// Find continuous sequence, whose sum is n
//////////////////////////////////////////////////////////////////////
////
void FindContinuousSequence(int n)
{
    if(n < 3)
        return;

    int small = 1;
    int big = 2;
    int middle = (1 + n) / 2;
    int sum = small + big;

    while(small < middle)
    {
        // we are lucky and find the sequence
        if(sum == n)
            PrintContinuousSequence(small, big);
```

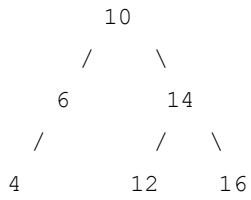
```
        // if the current sum is greater than n,
        // move small forward
        while(sum > n)
        {
            sum -= small;
            small ++;

            // we are lucky and find the sequence
            if(sum == n)
                PrintContinuousSequence(small, big);
        }

        // move big forward
        big ++;
        sum += big;
    }
}

////////////////////////////////////////////////////////////////////////
////
// Print continuous sequence between small and big
////////////////////////////////////////////////////////////////////////
////
void PrintContinuousSequence(int small, int big)
{
    for(int i = small; i <= big; ++ i)
        printf("%d ", i);

    printf("\n");
}
```

---

(27)-二元树的深度

题目：输入一棵二元树的根结点，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

例如：输入二元树：



输出该树的深度 3。

二元树的结点定义如下：

```
struct SBinaryTreeNode // a node of the binary tree
{
    int             m_nValue; // value of node
    SBinaryTreeNode  *m_pLeft;  // left child of node
    SBinaryTreeNode  *m_pRight; // right child of node
};
```

分析：这道题本质上还是考查二元树的遍历。

题目给出了一种树的深度的定义。当然，我们可以按照这种定义去得到树的所有路径，也就能得到最长路径以及它的长度。只是这种思路用来写程序有点麻烦。

我们还可以从另外一个角度来理解树的深度。如果一棵树只有一个结点，它的深度为 1。如果根结点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加 1；同样如果根结点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加 1。如果既有右子树又有左子树呢？那该树的深度就是其左、右子树深度的较大值再加 1。

上面的这个思路用递归的方法很容易实现，只需要对遍历的代码稍作修改即可。参考代码如下：

```
//////////////////////////////////////////////////////////////////////
//
// Get depth of a binary tree
// Input: pTreeNode - the head of a binary tree
// Output: the depth of a binary tree
//////////////////////////////////////////////////////////////////////
//
int TreeDepth(SBinaryTreeNode *pTreeNode)
{
    // the depth of a empty tree is 0
    if(!pTreeNode)
        return 0;

    // the depth of left sub-tree
    int nLeft = TreeDepth(pTreeNode->m_pLeft);
    // the depth of right sub-tree
```

```
    int nRight = TreeDepth(pTreeNode->m_pRight);

    // depth is the binary tree
    return (nLeft > nRight) ? (nLeft + 1) : (nRight + 1);
}
```

---

(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  PS:改变字符串顺序后必须还原回来!
            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。

---

(29)-调整数组顺序使奇数位于偶数前面

题目：输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。要求时间复杂度为 O(n)。

分析：如果不考虑时间复杂度，最简单的思路应该是从头扫描这个数组，每碰到一个偶数时，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该偶数放入这个空位。由于碰到一个偶数，需要移动 O(n)个数字，因此总的时间复杂度是 $O(n^2)$。

要求的是把奇数放在数组的前半部分，偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说我们在扫描这个数组的时候，如果发现有偶数出现在奇数的前面，我们可以交换他们的顺序，交换之后就符合要求了。

因此我们可以维护两个指针，第一个指针初始化为数组的第一个数字，它只向后移动；第二个指针初始化为数组的最后一个数字，它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶数而第二个指针指向的数字是奇数，我们就交换这两个数字。

基于这个思路，我们可以写出如下的代码：

```
void Reorder(int *pData, unsigned int length, bool (*func)(int));
bool isEven(int n);

////////////////////////////////////////////////////////////////////
////
// Devide an array of integers into two parts, odd in the first part,
// and even in the second part
// Input: pData  - an array of integers
//        length - the length of array
////////////////////////////////////////////////////////////////////
////
void ReorderOddEven(int *pData, unsigned int length)
{
    if(pData == NULL || length == 0)
        return;

    Reorder(pData, length, isEven);
}
```

```
////////////////////////////////////////////////////////////////////////
////
// Devide an array of integers into two parts, the intergers which
// satisfy func in the first part, otherwise in the second part
// Input: pData  - an array of integers
//        length - the length of array
//        func   - a function
////////////////////////////////////////////////////////////////////////
////
void Reorder(int *pData, unsigned int length, bool (*func)(int))
{
    if(pData == NULL || length == 0)
        return;

    int *pBegin = pData;
    int *pEnd = pData + length - 1;

    while(pBegin < pEnd)
    {
        // if *pBegin does not satisfy func, move forward
        if(!func(*pBegin))
        {
            pBegin ++;
            continue;
        }

        // if *pEnd does not satisfy func, move backward
        if(func(*pEnd))
        {
            pEnd --;
            continue;
        }

        // if *pBegin satisfy func while *pEnd does not,
        // swap these integers
        int temp = *pBegin;
        *pBegin = *pEnd;
        *pEnd = temp;
    }
}

////////////////////////////////////////////////////////////////////////
////
```

```
// Determine whether an integer is even or not
// Input: an integer
// otherwise return false
////////////////////////////////////////////////////////////////////////
////
bool isEven(int n)
{
    return (n & 1) == 0;
}
```

讨论：

上面的代码有三点值得提出来和大家讨论：

1．函数 isEven 判断一个数字是不是偶数并没有用%运算符而是用&。理由是通常情况下位运算符比%要快一些；

2．这道题有很多变种。这里要求是把奇数放在偶数的前面，如果把要求改成：把负数放在非负数的前面等，思路都是都一样的。

3．在函数 Reorder 中，用函数指针 func 指向的函数来判断一个数字是不是符合给定的条件，而不是用在代码直接判断（hard code）。这样的好处是把调整顺序的算法和调整的标准分开了（即解耦，decouple）。当调整的标准改变时，Reorder 的代码不需要修改，只需要提供一个新的确定调整标准的函数即可，提高了代码的可维护性。例如要求把负数放在非负数的前面，我们不需要修改 Reorder 的代码，只需添加一个函数来判断整数是不是非负数。这样的思路在很多库中都有广泛的应用，比如在ＳＴＬ的很多算法函数中都有一个仿函数（functor）的参数（当然仿函数不是函数指针，但其思想是一样的）。如果在面试中能够想到这一层，无疑能给面试官留下很好的印象。

---

(30)-异常安全的赋值运算符重载函数

题目：类 CMyString 的声明如下：

```
class CMyString
{
public:
    CMyString(char* pData = NULL);
    CMyString(const CMyString& str);
    ~CMyString(void);
    CMyString& operator = (const CMyString& str);
```

```
private:
    char* m_pData;
};
```

请实现其赋值运算符的重载函数，要求异常安全，即当对一个对象进行赋值时发生异常，对象的状态不能改变。

分析：首先我们来看一般 C++教科书上给出的赋值运算符的重载函数：

```
CMyString& CMyString::operator =(const CMyString &str)
{
    if(this == &str)
        return *this;

    delete []m_pData;
    m_pData = NULL;

    m_pData = new char[strlen(str.m_pData) + 1];
    strcpy(m_pData, str.m_pData);

    return *this;
}
```

我们知道，在分配内存时有可能发生异常。当执行语句 `new char[strlen(str.m_pData) + 1]` 发生异常时，程序将从该赋值运算符的重载函数退出不再执行。注意到这个时候语句 `delete []m_pData` 已经执行了。也就是说赋值操作没有完成，但原来对象的状态已经改变。也就是说不满足题目的异常安全的要求。

为了满足异常安全这个要求，一个简单的办法是掉换 new、delete 的顺序。先把内存 new 出来用一个临时指针保存起来，只有这个语句正常执行完成之后再执行 delete。这样就能够保证异常安全了。

下面给出的是一个更加优雅的实现方案：

```
CMyString& CMyString::operator =(const CMyString &str)
{
    if(this != &str)
    {
        CMyString strTemp(str);

        char* pTemp = strTemp.m_pData;
        strTemp.m_pData = m_pData;
        m_pData = pTemp;
    }
```

```
    return *this;
}
```

该方案通过调用构造拷贝函数创建一个临时对象来分配内存。此时即使发生异常，对原来对象的状态没有影响。交换临时对象和需要赋值的对象的字符串指针之后，由于临时对象的生命周期结束，自动调用其析构函数释放需赋值对象的原来的字符串空间。整个函数不需要显式用到 new、delete，内存的分配和释放都自动完成，因此代码显得比较优雅。

---

(31)-从尾到头输出链表

题目：输入一个链表的头结点，从尾到头反过来输出每个结点的值。链表结点定义如下：

```
struct ListNode
{
    int      m_nKey;
    ListNode* m_pNext;
};
```

分析：这是一道很有意思的面试题。该题以及它的变体经常出现在各大公司的面试、笔试题中。

看到这道题后，第一反应是从头到尾输出比较简单。于是很自然地想到把链表中链接结点的指针反转过来，改变链表的方向。然后就可以从头到尾输出了。反转链表的算法详见本人面试题精选系列的第 19 题，在此不再细述。但该方法需要额外的操作，应该还有更好的方法。

接下来的想法是从头到尾遍历链表，每经过一个结点的时候，把该结点放到一个栈中。当遍历完整个链表后，再从栈顶开始输出结点的值，此时输出的结点的顺序已经反转过来了。该方法需要维护一个额外的栈，实现起来比较麻烦。

既然想到了栈来实现这个函数，而递归本质上就是一个栈结构。于是很自然的又想到了用递归来实现。要实现反过来输出链表，我们每访问到一个结点的时候，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。

基于这样的思路，不难写出如下代码：

```
//////////////////////////////////////////////////////////////////////
//
// Print a list from end to beginning
// Input: pListHead - the head of list
```

```
//////////////////////////////////////////////////////////////////////
//
void PrintListReversely(ListNode* pListHead)
{
    if(pListHead != NULL)
    {
        // Print the next node first
        if (pListHead->m_pNext != NULL)
        {
            PrintListReversely(pListHead->m_pNext);
        }

        // Print this node
        printf("%d", pListHead->m_nKey);
    }
}
```

扩展：该题还有两个常见的变体：

1. 从尾到头输出一个字符串；
2. 定义一个函数求字符串的长度，要求该函数体内不能声明任何变量。

---

(32)-不能被继承的类

题目：用 C++设计一个不能被继承的类。

分析：这是 Adobe 公司 2007 年校园招聘的最新笔试题。这道题除了考察应聘者的 C++基本功底外，还能考察反应能力，是一道很好的题目。

在 Java 中定义了关键字 final，被 final 修饰的类不能被继承。但在 C++中没有 final 这个关键字，要实现这个要求还是需要花费一些精力。

首先想到的是在 C++ 中，子类的构造函数会自动调用父类的构造函数。同样，子类的析构函数也会自动调用父类的析构函数。要想一个类不能被继承，我们只要把它的构造函数和析构函数都定义为私有函数。那么当一个类试图从它那继承的时候，必然会由于试图调用构造函数、析构函数而导致编译错误。

可是这个类的构造函数和析构函数都是私有函数了，我们怎样才能得到该类的实例呢？这难不倒我们，我们可以通过定义静态来创建和释放类的实例。基于这个思路，我们可以写出如下的代码：

```
//////////////////////////////////////////////////////////////////////
//
// Define a class which can't be derived from
//////////////////////////////////////////////////////////////////////
//
class FinalClass1
{
public:
    static FinalClass1* GetInstance()
    {
        return new FinalClass1;
    }

    static void DeleteInstance( FinalClass1* pInstance)
    {
        delete pInstance;
        pInstance = 0;
    }

private:
    FinalClass1() {}
    ~FinalClass1() {}
};
```

这个类是不能被继承，但在总觉得它和一般的类有些不一样，使用起来也有点不方便。比如，我们只能得到位于堆上的实例，而得不到位于栈上实例。

能不能实现一个和一般类除了不能被继承之外其他用法都一样的类呢？办法总是有的，不过需要一些技巧。请看如下代码：

```
//////////////////////////////////////////////////////////////////////
//
// Define a class which can't be derived from
//////////////////////////////////////////////////////////////////////
//
template <typename T> class MakeFinal
{
    friend T;

private:
    MakeFinal() {}
    ~MakeFinal() {}
};

class FinalClass2 : virtual public MakeFinal<FinalClass2>
```

```
{
public:
    FinalClass2() {}
    ~FinalClass2() {}
};
```

这个类使用起来和一般的类没有区别，可以在栈上、也可以在堆上创建实例。尽管类 MakeFinal<FinalClass2>的构造函数和析构函数都是私有的，但由于类 FinalClass2 是它的友元函数，因此在 FinalClass2 中调用 MakeFinal<FinalClass2>的构造函数和析构函数都不会造成编译错误。

但当我们试图从 FinalClass2 继承一个类并创建它的实例时，却不同通过编译。

```
class Try : public FinalClass2
{
public:
    Try() {}
    ~Try() {}
};

Try temp;
```

由于类 FinalClass2 是从类 MakeFinal<FinalClass2>虚继承过来的，在调用 Try 的构造函数的时候，会直接跳过 FinalClass2 而直接调用 MakeFinal<FinalClass2> 的构造函数。非常遗憾的是，Try 不是 MakeFinal<FinalClass2>的友元，因此不能调用其私有的构造函数。

基于上面的分析，试图从 FinalClass2 继承的类，一旦实例化，都会导致编译错误，因此是 FinalClass2 不能被继承。这就满足了我们设计要求。

---

(33)-在 O(1)时间删除链表结点

题目：给定链表的头指针和一个结点指针，在 O(1)时间删除该结点。链表结点的定义如下：

```
struct ListNode
{
    int       m_nKey;
    ListNode*  m_pNext;
};
```

函数的声明如下：

```
void DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted);
```

分析：这是一道广为流传的 Google 面试题，能有效考察我们的编程基本功，还能考察我们的反应速度，更重要的是，还能考察我们对时间复杂度的理解。

在链表中删除一个结点，最常规的做法是从链表的头结点开始，顺序查找要删除的结点，找到之后再删除。由于需要顺序查找，时间复杂度自然就是 O(n) 了。

我们之所以需要从头结点开始查找要删除的结点，是因为我们需要得到要删除的结点的前面一个结点。我们试着换一种思路。我们可以从给定的结点得到它的下一个结点。这个时候我们实际删除的是它的下一个结点，由于我们已经得到实际删除的结点的前面一个结点，因此完全是可以实现的。当然，在删除之前，我们需要需要把给定的结点的下一个结点的数据拷贝到给定的结点中。此时，时间复杂度为 O(1)。

上面的思路还有一个问题：如果删除的结点位于链表的尾部，没有下一个结点，怎么办？我们仍然从链表的头结点开始，顺便遍历得到给定结点的前序结点，并完成删除操作。这个时候时间复杂度是 O(n)。

那题目要求我们需要在 O(1)时间完成删除操作，我们的算法是不是不符合要求？实际上，假设链表总共有 n 个结点，我们的算法在 n-1 总情况下时间复杂度是 O(1)，只有当给定的结点处于链表末尾的时候，时间复杂度为 O(n)。那么平均时间复杂度[(n-1)*O(1)+O(n)]/n，仍然为 O(1)。

基于前面的分析，我们不难写出下面的代码。

参考代码：

```
//////////////////////////////////////////////////////////////////////
//
// Delete a node in a list
// Input: pListHead - the head of list
//        pToBeDeleted - the node to be deleted
//////////////////////////////////////////////////////////////////////
//
void DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted)
{
    if(!pListHead || !pToBeDeleted)
        return;

    // if pToBeDeleted is not the last node in the list
    if(pToBeDeleted->m_pNext != NULL)
    {
        // copy data from the node next to pToBeDeleted
        ListNode* pNext = pToBeDeleted->m_pNext;
        pToBeDeleted->m_nKey = pNext->m_nKey;
        pToBeDeleted->m_pNext = pNext->m_pNext;
```

```
        // delete the node next to the pToBeDeleted
        delete pNext;
        pNext = NULL;
    }
    // if pToBeDeleted is the last node in the list
    else
    {
        // get the node prior to pToBeDeleted
        ListNode* pNode = pListHead;
        while(pNode->m_pNext != pToBeDeleted)
        {
            pNode = pNode->m_pNext;
        }

        // deleted pToBeDeleted
        pNode->m_pNext = NULL;
        delete pToBeDeleted;
        pToBeDeleted = NULL;
    }
}
```

值得注意的是，为了让代码看起来简洁一些，上面的代码基于两个假设：（1）给定的结点的确在链表中；（2）给定的要删除的结点不是链表的头结点。不考虑第一个假设对代码的鲁棒性是有影响的。至于第二个假设，当整个列表只有一个结点时，代码会有问题。但这个假设不算很过分，因为在有些链表的实现中，会创建一个虚拟的链表头，并不是一个实际的链表结点。这样要删除的结点就不可能是链表的头结点了。当然，在面试中，我们可以把这些假设和面试官交流。这样，面试官还是会觉得我们考虑问题很周到的。

PS:假设要删除 A 节点，A.next==B。则删除 B，并将 B 的值赋给 A，这样就等于删除了 A 节点。

---

(34)-找出数组中两个只出现一次的数字

题目：一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是 O(n)，空间复杂度是 O(1)。

分析：这是一道很新颖的关于位运算的面试题。

首先我们考虑这个问题的一个简单版本：一个数组里除了一个数字之外，其他的数字都出现了两次。请写程序找出这个只出现一次的数字。

这个题目的突破口在哪里？题目为什么要强调有一个数字出现一次，其他的出现两次？我们想到了异或运算的性质：任何一个数字异或它自己都等于 0。也就是说，如果我们从头到尾依次异或数组中的每一个数字，那么最终的结果刚好是那个只出现依次的数字，因为那些出现两次的数字全部在异或中抵消掉了。

有了上面简单问题的解决方案之后，我们回到原始的问题。如果能够把原数组分为两个子数组。在每个子数组中，包含一个只出现一次的数字，而其他数字都出现两次。如果能够这样拆分原数组，按照前面的办法就是分别求出这两个只出现一次的数字了。

我们还是从头到尾依次异或数组中的每一个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果。因为其他数字都出现了两次，在异或中全部抵消掉了。由于这两个数字肯定不一样，那么这个异或结果肯定不为 0，也就是说在这个结果数字的二进制表示中至少就有一位为 1。我们在结果数字中找到第一个为 1 的位的位置，记为第 N 位。现在我们以第 N 位是不是 1 为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第 N 位都为 1，而第二个子数组的每个数字的第 N 位都为 0。

现在我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。因此到此为止，所有的问题我们都已经解决。

基于上述思路，我们不难写出如下代码：

```
//////////////////////////////////////////////////////////////////////
//
// Find two numbers which only appear once in an array
// Input: data - an array contains two number appearing exactly once,
//             while others appearing exactly twice
//       length - the length of data
// Output: num1 - the first number appearing once in data
//        num2 - the second number appearing once in data
//////////////////////////////////////////////////////////////////////
//
void FindNumsAppearOnce(int data[], int length, int &num1, int &num2)
{
    if (length < 2)
        return;

    // get num1 ^ num2
    int resultExclusiveOR = 0;
    for (int i = 0; i < length; ++ i)
        resultExclusiveOR ^= data[i];

    // get index of the first bit, which is 1 in resultExclusiveOR
```

```
    unsigned int indexOf1 = FindFirstBitIs1(resultExclusiveOR);

    num1 = num2 = 0;
    for (int j = 0; j < length; ++ j)
    {
        // divide the numbers in data into two groups,
        // the indexOf1 bit of numbers in the first group is 1,
        // while in the second group is 0
        if(IsBit1(data[j], indexOf1))
            num1 ^= data[j];
        else
            num2 ^= data[j];
    }
}

//////////////////////////////////////////////////////////////////////
//
// Find the index of first bit which is 1 in num (assuming not 0)
//////////////////////////////////////////////////////////////////////
//
unsigned int FindFirstBitIs1(int num)
{
    int indexBit = 0;
    while (((num & 1) == 0) && (indexBit < 32))
    {
        num = num >> 1;
        ++ indexBit;
    }

    return indexBit;
}

//////////////////////////////////////////////////////////////////////
//
// Is the indexBit bit of num 1?
//////////////////////////////////////////////////////////////////////
//
bool IsBit1(int num, unsigned int indexBit)
{
    num = num >> indexBit;

    return (num & 1);
}
```

PS:关键词->相同数字异或结果为 0，并根据所有元素异或结果第一个出现 0 的位置把数组分为 2 部分！！！

---

(35)-找出两个链表的第一个公共结点

题目：两个单向链表，找出它们的第一个公共结点。

链表的结点定义为：

```
struct ListNode
{
    int      m_nKey;
    ListNode*   m_pNext;
};
```

分析：这是一道微软的面试题。微软非常喜欢与链表相关的题目，因此在微软的面试题中，链表出现的概率相当高。

如果两个单向链表有公共的结点，也就是说两个链表从某一结点开始，它们的 m_pNext 都指向同一个结点。但由于是单向链表的结点，每个结点只有一个 m_pNext，因此从第一个公共结点开始，之后它们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个 Y，而不可能像 X。

看到这个题目，第一反应就是蛮力法：在第一链表上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们的公共结点。如果第一个链表的长度为 m，第二个链表的长度为 n，显然，该方法的时间复杂度为 O(mn)。

接下来我们试着去寻找一个线性时间复杂度的算法。我们先把问题简化：如何判断两个单向链表有没有公共结点？前面已经提到，如果两个链表有一个公共结点，那么该公共结点之后的所有结点都是重合的。那么，它们的最后一个结点必然是重合的。因此，我们判断两个链表是不是有重合的部分，只要分别遍历两个链表到最后一个结点。如果两个尾结点是一样的，说明它们用重合；否则两个链表没有公共的结点。

在上面的思路中，顺序遍历两个链表到尾结点的时候，我们不能保证在两个链表上同时到达尾结点。这是因为两个链表不一定长度一样。但如果假设一个链表比另一个长 l 个结点，我们先在长的链表上遍历 l 个结点，之后再同步遍历，这个时候我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点考试到链表的尾结点，这一部分是重合的。因此，它们肯定也是同时到达第一公共结点的。于是在遍历中，第一个相同的结点就是第一个公共的结点。

在这个思路中，我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，知道找到相同的结点，或者一直到链表结束。此时，如果第一个链表的长度为 m，第二个链表的长度为 n，该方法的时间复杂度为 O(m+n)。

基于这个思路，我们不难写出如下的代码：

```
////////////////////////////////////////////////////////////////////
//
// Find the first common node in the list with head pHead1 and
// the list with head pHead2
// Input: pHead1 - the head of the first list
//       pHead2 - the head of the second list
// Return: the first common node in two list. If there is no common
//        nodes, return NULL
////////////////////////////////////////////////////////////////////
//
ListNode* FindFirstCommonNode( ListNode *pHead1, ListNode *pHead2)
{
    // Get the length of two lists
    unsigned int nLength1 = ListLength(pHead1);
    unsigned int nLength2 = ListLength(pHead2);
    int nLengthDif = nLength1 - nLength2;

    // Get the longer list
    ListNode *pListHeadLong = pHead1;
    ListNode *pListHeadShort = pHead2;
    if(nLength2 > nLength1)
    {
        pListHeadLong = pHead2;
        pListHeadShort = pHead1;
        nLengthDif = nLength2 - nLength1;
    }

    // Move on the longer list
    for(int i = 0; i < nLengthDif; ++ i)
        pListHeadLong = pListHeadLong->m_pNext;

    // Move on both lists
    while((pListHeadLong != NULL) &&
        (pListHeadShort != NULL) &&
        (pListHeadLong != pListHeadShort))
    {
        pListHeadLong = pListHeadLong->m_pNext;
        pListHeadShort = pListHeadShort->m_pNext;
```

```
    }

    // Get the first common node in two lists
    ListNode *pFisrtCommonNode = NULL;
    if(pListHeadLong == pListHeadShort)
        pFisrtCommonNode = pListHeadLong;

    return pFisrtCommonNode;
}

//////////////////////////////////////////////////////////////////////
//
// Get the length of list with head pHead
// Input: pHead - the head of list
// Return: the length of list
//////////////////////////////////////////////////////////////////////
//
unsigned int ListLength(ListNode* pHead)
{
    unsigned int nLength = 0;
    ListNode* pNode = pHead;
    while(pNode != NULL)
    {
        ++ nLength;
        pNode = pNode->m_pNext;
    }

    return nLength;
}
```

PS:两个有公共结点而部分重合的链表，拓扑形状看起来像一个 Y，而不可能像 X。即从公共节点开始之后的所有都相同！求出链表长度差，遍历差数目长的链表，之后同步遍历两链表，第一个相同的节点即是结果。

---

(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 $O(n^2)$。

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 2^8=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';
}
```

PS：删除字符：使用两个临时指针变量。
查找字符串：hash

(37)-寻找丑数

题目：我们把只包含因子
2、3 和 5 的数称作丑数（Ugly Number）。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。习惯上我们把 1 当做是第一个丑数。求按从小到大的顺序的第 1500 个丑数。
分析：这是一道在网络上广为流传的面试题，据说 google 曾经采用过这道题。

所谓一个数 m 是另一个数 n 的因子，是指 n 能被 m 整除，也就是 n % m == 0。根据丑数的定义，丑数只能被 2、3 和 5 整除。也就是说如果一个数如果它能被 2 整除，我们把它连续除以 2；如果能被 3 整除，就连续除以 3；如果能被 5 整除，就除以连续 5。如果最后我们得到的是 1，那么这个数就是丑数，否则不是。

基于前面的分析，我们可以写出如下的函数来判断一个数是不是丑数：

```
bool IsUgly(int number)
{
    while(number % 2 == 0)
        number /= 2;
    while(number % 3 == 0)
        number /= 3;
    while(number % 5 == 0)
        number /= 5;

    return (number == 1) ? true : false;
}
```

接下来，我们只需要按顺序判断每一个整数是不是丑数，即：

```
int GetUglyNumber_Solution1(int index)
{
    if(index <= 0)
        return 0;

    int number = 0;
    int uglyFound = 0;
    while(uglyFound < index)
    {
        ++number;

        if(IsUgly(number))
        {
            ++uglyFound;
        }
    }

    return number;
}
```

我们只需要在函数 GetUglyNumber_Solution1 中传入参数 1500，就能得到第 1500 个丑数。该算法非常直观，代码也非常简洁，但最大的问题我们每个整数都需要计算。即使一个数字不是丑数，我们还是需要对它做求余数和除法操作。因此该算法的时间效率不是很高。

接下来我们换一种思路来分析这个问题，试图只计算丑数，而不在非丑数的整数上花费时间。根据丑数的定义，丑数应该是另一个丑数乘以 2、3 或者 5 的结果（1 除外）。因此我们可以创建一个数组，里面的数字是排好序的丑数。里面的每一个丑数是前面的丑数乘以 2、3 或者 5 得到的。

这种思路的关键在于怎样确保数组里面的丑数是排好序的。我们假设数组中已经有若干个丑数，排好序后存在数组中。我们把现有的最大丑数记做 M。现在我们来生成下一个丑数，该丑数肯定是前面某一个丑数乘以 2、3 或者 5 的结果。我们首先考虑把已有的每个丑数乘以 2。在乘以 2 的时候，能得到若干个结果小于或等于 M 的。由于我们是按照顺序生成的，小于或者等于 M 肯定已经在数组中了，我们不需再次考虑；我们还会得到若干个大于 M 的结果，但我们只需要第一个大于 M 的结果，因为我们希望丑数是按从小到大顺序生成的，其他更大的结果我们以后再说。我们把得到的第一个乘以 2 后大于 M 的结果，记为 M_2。同样我们把已有的每一个丑数乘以 3 和 5，能得到第一个大于 M 的结果 M_3 和 M_5。那么下一个丑数应该是 M_2、M_3 和 M_5 三个数的最小者。

前面我们分析的时候，提到把已有的每个丑数分别都乘以 2、3 和 5，事实上是不需要的，因为已有的丑数是按顺序存在数组中的。对乘以 2 而言，肯定存在某一个丑数 T_2，排在它之前的每一个丑数乘以 2 得到的结果都会小于已有最大的丑数，在它之后的每一个丑数乘以 2 得到的结果都会太大。我们只需要记下这个丑数的位置，同时每次生成新的丑数的时候，去更新这个 T_2。对乘以 3 和 5 而言，存在着同样的 T_3 和 T_5。

有了这些分析，我们不难写出如下的代码：

```
int GetUglyNumber_Solution2(int index)
{
    if(index <= 0)
        return 0;

    int *pUglyNumbers = new int[index];
    pUglyNumbers[0] = 1;
    int nextUglyIndex = 1;

    int *pMultiply2 = pUglyNumbers;
    int *pMultiply3 = pUglyNumbers;
    int *pMultiply5 = pUglyNumbers;

    while(nextUglyIndex < index)
    {
        int min = Min(*pMultiply2 * 2, *pMultiply3 * 3, *pMultiply5 * 5);
        pUglyNumbers[nextUglyIndex] = min;

        while(*pMultiply2 * 2 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply2;
        while(*pMultiply3 * 3 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply3;
        while(*pMultiply5 * 5 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply5;
```

```
            ++nextUglyIndex;
        }

        int ugly = pUglyNumbers[nextUglyIndex - 1];
        delete[] pUglyNumbers;
        return ugly;
    }

    int Min(int number1, int number2, int number3)
    {
        int min = (number1 < number2) ? number1 : number2;
        min = (min < number3) ? min : number3;

        return min;
    }
```

和第一种思路相比，这种算法不需要在非丑数的整数上做任何计算，因此时间复杂度要低很多。感兴趣的读者可以分别统计两个函数 GetUglyNumber_Solution1(1500) 和 GetUglyNumber_Solution2(1500)的运行时间。当然我们也要指出，第二种算法由于要保存已经生成的丑数，因此需要一个数组，从而需要额外的内存。第一种算法是没有这样的内存开销的。

(38)-输出 1 到最大的 N 位数

题目：输入数字 n，按顺序输出从 1 最大的 n 位 10 进制数。比如输入 3，则输出 1、2、3 一直到最大的 3 位数即 999。

分析：这是一道很有意思的题目。看起来很简单，其实里面却有不少的玄机。

应聘者在解决这个问题的时候，最容易想到的方法是先求出最大的 n 位数是什么，然后用一个循环从 1 开始逐个输出。很快，我们就能写出如下代码：

```
    // Print numbers from 1 to the maximum number with n digits, in order
    void Print1ToMaxOfNDigits_1(int n)
    {
        // calculate 10^n
        int number = 1;
        int i = 0;
        while(i++ < n)
            number *= 10;

        // print from 1 to (10^n - 1)
        for(i = 1; i < number; ++i)
            printf("%d\t", i);
    }
```

初看之下，好像没有问题。但如果我们仔细分析这个问题，就能注意到这里没有规定 n 的范围，当我们求最大的 n 位数的时候，是不是有可能用整型甚至长整型都会溢出？

分析到这里，我们很自然的就想到我们需要表达一个大数。最常用的也是最容易实现的表达大数的方法是用字符串或者整型数组（当然不一定是最有效的）。

用字符串表达数字的时候，最直观的方法就是字符串里每个字符都是’0’到’9’之间的某一个字符，表示数字中的某一位。因为数字最大是 n 位的，因此我们需要一个 n+1 位字符串（最后一位为结束符号’\0’）。当实际数字不够 n 位的时候，在字符串的前半部分补零。这样，数字的个位永远都在字符串的末尾（除去结尾符号）。

首先我们把字符串中每一位数字都初始化为’0’。然后每一次对字符串表达的数字加 1，再输出。因此我们只需要做两件事：一是在字符串表达的数字上模拟加法。另外我们要把字符串表达的数字输出。值得注意的是，当数字不够 n 位的时候，我们在数字的前面补零。输出的时候这些补位的 0 不应该输出。比如输入 3 的时候，那么数字 98 以 098 的形式输出，就不符合我们的习惯了。

基于上述分析，我们可以写出如下代码：

```
// Print numbers from 1 to the maximum number with n digits, in order
void Print1ToMaxOfNDigits_2(int n)
{
    // 0 or minus numbers are invalid input
    if(n <= 0)
        return;

    // number is initialized as 0
    char *number = new char[n + 1];
    memset(number, '0', n);
    number[n] = '\0';

    while(!Increment(number))
    {
        PrintNumber(number);
    }

    delete []number;
}

// Increment a number. When overflow, return true; otherwise return false
bool Increment(char* number)
{
    bool isOverflow = false;
    int nTakeOver = 0;
    int nLength = strlen(number);

    // Increment (Add 1) operation begins from the end of number
    for(int i = nLength - 1; i >= 0; i --)
    {
        int nSum = number[i] - '0' + nTakeOver;
```

```
        if(i == nLength - 1)
            nSum ++;

        if(nSum >= 10)
        {
            if(i == 0)
                isOverflow = true;
            else
            {
                nSum -= 10;
                nTakeOver = 1;
                number[i] = '0' + nSum;
            }
        }
        else
        {
            number[i] = '0' + nSum;
            break;
        }
    }

    return isOverflow;
}

// Print number stored in string, ignore 0 at its beginning
// For example, print "0098" as "98"
void PrintNumber(char* number)
{
    bool isBeginning0 = true;
    int nLength = strlen(number);

    for(int i = 0; i < nLength; ++ i)
    {
        if(isBeginning0 && number[i] != '0')
            isBeginning0 = false;

        if(!isBeginning0)
        {
            printf("%c", number[i]);
        }
    }

    printf("\t");
}
```

第二种思路基本上和第一种思路相对应，只是把一个整型数值换成了字符串的表示形式。同时，值得提出的是，判断打印是否应该结束时，我没有调用函数 strcmp 比较字符串 number 和”99…999”（n 个 9）。这是因为 strcmp 的时间复杂度是 O(n)，而判断是否溢出的平均时间复杂度是 O(1)。

第二种思路虽然比较直观，但由于模拟了整数的加法，代码有点长。要在面试短短几十分钟时间里完整正确写出这么长代码，不是件容易的事情。接下来我们换一种思路来考虑这个问题。如果我们在数字前面补 0 的话，就会发现 n 位所有 10 进制数其实就是 n 个从 0 到 9 的全排列。也就是说，我们把数字的每一位都从 0 到 9 排列一遍，就得到了所有的 10 进制数。只是我们在输出的时候，数字排在前面的 0 我们不输出罢了。

全排列用递归很容易表达，数字的每一位都可能是 0 到 9 中的一个数，然后设置下一位。递归结束的条件是我们已经设置了数字的最后一位。

```
// Print numbers from 1 to the maximum number with n digits, in order
void Print1ToMaxOfNDigits_3(int n)
{
    // 0 or minus numbers are invalid input
    if(n <= 0)
        return;

    char* number = new char[n + 1];
    number[n] = '\0';

    for(int i = 0; i < 10; ++i)
    {
        // first digit can be 0 to 9
        number[0] = i + '0';

        Print1ToMaxOfNDigitsRecursively(number, n, 0);
    }

    delete[] number;
}

// length: length of number
// index: current index of digit in number for this round
void Print1ToMaxOfNDigitsRecursively(char* number, int length, int index)
{
    // we have reached the end of number, print it
    if(index == length - 1)
    {
        PrintNumber(number);
        return;
    }

    for(int i = 0; i < 10; ++i)
    {
```

```
            // next digit can be 0 to 9
            number[index + 1] = i + '0';

            // go to the next digit
            Print1ToMaxOfNDigitsRecursively(number, length, index + 1);
        }
    }
```

函数 PrintNumber 和前面第二种思路中的一样，这里就不重复了。对比这两种思路，我们可以发现，递归能够用很简洁的代码来解决问题。

---

(39)-颠倒栈

题目：用递归颠倒一个栈。例如输入栈{1, 2, 3, 4, 5}，1 在栈顶。颠倒之后的栈为{5, 4, 3, 2, 1}，5 处在栈顶。

分析：乍一看到这道题目，第一反应是把栈里的所有元素逐一 pop 出来，放到一个数组里，然后在数组里颠倒所有元素，最后把数组中的所有元素逐一 push 进入栈。这时栈也就颠倒过来了。颠倒一个数组是一件很容易的事情。不过这种思路需要显示分配一个长度为 O(n)的数组，而且也没有充分利用递归的特性。

我们再来考虑怎么递归。我们把栈{1, 2, 3, 4, 5}看成由两部分组成：栈顶元素 1 和剩下的部分{2, 3, 4, 5}。如果我们能把{2, 3, 4, 5}颠倒过来，变成{5, 4, 3, 2}，然后在把原来的栈顶元素 1 放到底部，那么就整个栈就颠倒过来了，变成{5, 4, 3, 2, 1}。

接下来我们需要考虑两件事情：一是如何把{2, 3, 4, 5}颠倒过来变成{5, 4, 3, 2}。我们只要把{2, 3, 4, 5}看成由两部分组成：栈顶元素 2 和剩下的部分{3, 4, 5}。我们只要把{3, 4, 5}先颠倒过来变成{5, 4, 3}，然后再把之前的栈顶元素 2 放到最底部，也就变成了{5, 4, 3, 2}。

至于怎么把{3, 4, 5}颠倒过来……很多读者可能都想到这就是递归。也就是每一次试图颠倒一个栈的时候，现在栈顶元素 pop 出来，再颠倒剩下的元素组成的栈，最后把之前的栈顶元素放到剩下元素组成的栈的底部。递归结束的条件是剩下的栈已经空了。这种思路的代码如下：

```
// Reverse a stack recursively in three steps:
// 1. Pop the top element
// 2. Reverse the remaining stack
// 3. Add the top element to the bottom of the remaining stack
template<typename T> void ReverseStack(std::stack<T>& stack)
{
    if(!stack.empty())
    {
        T top = stack.top();
        stack.pop();
        ReverseStack(stack);
        AddToStackBottom(stack, top);
    }
}
```

我们需要考虑的另外一件事情是如何把一个元素 *e* 放到一个栈的底部，也就是如何实现 AddToStackBottom。这件事情不难，只需要把栈里原有的元素逐一 pop 出来。当栈为空的时候，push 元素 *e* 进栈，此时它就位于栈的底部了。然后再把栈里原有的元素按照 pop 相反的顺序逐一 push 进栈。

注意到我们在 push 元素 *e* 之前，我们已经把栈里原有的所有元素都 pop 出来了，我们需要把它们保存起来，以便之后能把他们再 push 回去。我们当然可以开辟一个数组来做，但这没有必要。由于我们可以用递归来做这件事情，而递归本身就是一个栈结构。我们可以用递归的栈来保存这些元素。

基于如上分析，我们可以写出 AddToStackBottom 的代码：

```
// Add an element to the bottom of a stack:
template<typename T> void AddToStackBottom(std::stack<T>& stack, T t)
{
    if(stack.empty())
    {
        stack.push(t);
    }
    else
    {
        T top = stack.top();
        stack.pop();
        AddToStackBottom(stack, t);
        stack.push(top);
    }
}
```

---

(40)-扑克牌的顺子

题目：从扑克牌中随机抽 5 张牌，判断是不是一个顺子，即这 5 张牌是不是连续的。2-10 为数字本身，A 为 1，J 为 11，Q 为 12，K 为 13，而大小王可以看成任意数字。

分析：这题目很有意思，是一个典型的寓教于乐的题目。

我们需要把扑克牌的背景抽象成计算机语言。不难想象，我们可以把 5 张牌看成由 5 个数字组成的数组。大小王是特殊的数字，我们不妨把它们都当成 0，这样和其他扑克牌代表的数字就不重复了。

接下来我们来分析怎样判断 5 个数字是不是连续的。最直观的是，我们把数组排序。但值得注意的是，由于 0 可以当成任意数字，我们可以用 0 去补满数组中的空缺。也就是排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，但如果我们有足够的 0 可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0，1，3，4，5}。在 1 和 3 之间空缺了一个 2，刚好我们有一个 0，也就是我们可以它当成 2 去填补这个空缺。

于是我们需要做三件事情：把数组排序，统计数组中 0 的个数，统计排序之后的数组相邻数字之间的空缺总数。如果空缺的总数小于或者等于 0 的个数，那么这个数组就是连续的；反之则不连续。最后，我们还需要注意的是，如果数组中的非 0 数字重复出现，则该数组不是连续的。换成扑克牌的描述方式，就是如果一副牌里含有对子，则不可能是顺子。

基于这个思路，我们可以写出如下的代码：

```
// Determine whether numbers in an array are continuous
// Parameters: numbers: an array, each number in the array is between
//             0 and maxNumber. 0 can be treeted as any number between
//             1 and maxNumber
//             maxNumber: the maximum number in the array numbers
bool IsContinuous(std::vector<int> numbers, int maxNumber)
{
    if(numbers.size() == 0 || maxNumber <=0)
        return false;

    // Sort the array numbers.
    std::sort(numbers.begin(), numbers.end());

    int numberOfZero = 0;
    int numberOfGap = 0;

    // how many 0s in the array?
    std::vector<int>::iterator smallerNumber = numbers.begin();
    while(smallerNumber != numbers.end() && *smallerNumber == 0)
    {
        numberOfZero++;
        ++smallerNumber;
    }

    // get the total gaps between all adjacent two numbers
    std::vector<int>::iterator biggerNumber = smallerNumber + 1;
    while(biggerNumber < numbers.end())
    {
        // if any non-zero number appears more than once in the array,
        // the array can't be continuous
        if(*biggerNumber == *smallerNumber)
            return false;

        numberOfGap += *biggerNumber - *smallerNumber - 1;
        smallerNumber = biggerNumber;
        ++biggerNumber;
    }

    return (numberOfGap > numberOfZero) ? false : true;
}
```

本文为了让代码显得比较简洁，上述代码用 C++的标准模板库中的 vector 来表达数组，同时用函数 sort 排序。当然我们可以自己写排序算法。为了有更好的通用性，上述代码没有限定数组的长度和允许出现的最大数字。要解答原题，我们只需要确保传入的数组的长度是 5，并且 maxNumber 为 13 即可。

(41)-把数组排成最小的数

题目：输入一个正整数数组，将它们连接起来排成一个数，输出能排出的所有数字中最小的一个。例如输入数组{32, 321}，则输出这两个能排成的最小数字 32132。请给出解决问题的算法，并证明该算法。

分析：这是 09 年 6 月份百度新鲜出炉的一道面试题，从这道题我们可以看出百度对应聘者在算法方面有很高的要求。

这道题其实是希望我们能找到一个排序规则，根据这个规则排出来的数组能排成一个最小的数字。要确定排序规则，就得比较两个数字，也就是给出两个数字 m 和 n，我们需要确定一个规则 m 和 n 哪个更大，而不是仅仅只是比较这两个数字的数值哪个更大。

根据题目的要求，两个数字 m 和 n 排成的数字 mn 和 nm，如果 mn<nm，那么我们应该输出 mn，也就是 m 应该排在 n 的前面，也就是 m 小于 n；反之，如果 nm<mn，n 小于 m。如果 mn==mn，m 等于 n。

接下来我们考虑怎么去拼接数字，即给出数字 m 和 n，怎么得到数字 mn 和 nm 并比较它们的大小。直接用数值去计算不难办到，但需要考虑到的一个潜在问题是 m 和 n 都在 int 能表达的范围内，但把它们拼起来的数字 mn 和 nm 就不一定能用 int 表示了。所以我们需要解决大数问题。一个非常直观的方法就是把数字转换成字符串。

另外，由于把数字 m 和 n 拼接起来得到的 mn 和 nm，它们所含有的数字的个数肯定是相同的。因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

基于这个思路，我们可以写出下面的代码：

```
// Maxinum int number has 10 digits in decimal system
const int g_MaxNumberLength = 10;

// String buffers to combine two numbers
char* g_StrCombine1 = new char[g_MaxNumberLength * 2 + 1];
char* g_StrCombine2 = new char[g_MaxNumberLength * 2 + 1];

// Given an array, print  the minimum number
// by combining all numbers in the array
void PrintMinNumber(int* numbers, int length)
{
    if(numbers == NULL || length <= 0)
        return;

    // Convert all numbers as strings
    char** strNumbers = (char**)(new int[length]);
    for(int i = 0; i < length; ++i)
    {
        strNumbers[i] = new char[g_MaxNumberLength + 1];
        sprintf(strNumbers[i], "%d", numbers[i]);
    }
```

```
    // Sort all strings according to algorithm in function compare
    qsort(strNumbers, length, sizeof(char*), compare);

    for(int i = 0; i < length; ++i)
        printf("%s", strNumbers[i]);
    printf("\n");

    for(int i = 0; i < length; ++i)
        delete[] strNumbers[i];
    delete[] strNumbers;
}

// Compare two numbers in strNumber1 and strNumber2
// if [strNumber1][strNumber2] > [strNumber2][strNumber1],
// return value > 0
// if [strNumber1][strNumber2] = [strNumber2][strNumber1],
// return value = 0
// if [strNumber1][strNumber2] < [strNumber2][strNumber1],
// return value < 0
int compare(const void* strNumber1, const void* strNumber2)
{
    // [strNumber1][strNumber2]
    strcpy(g_StrCombine1, *(const char**)strNumber1);
    strcat(g_StrCombine1, *(const char**)strNumber2);

    // [strNumber2][strNumber1]
    strcpy(g_StrCombine2, *(const char**)strNumber2);
    strcat(g_StrCombine2, *(const char**)strNumber1);

    return strcmp(g_StrCombine1, g_StrCombine2);
}
```

上述代码中，我们在函数 compare 中定义比较规则，并根据该规则用库函数 qsort 排序。最后把排好序的数组输出，就得到了根据数组排成的最小的数字。

找到一个算法解决这个问题，不是一件容易的事情。但更困难的是我们需要证明这个算法是正确的。接下来我们来试着证明。

首先我们需要证明之前定义的比较两个数字大小的规则是有效的。一个有效的比较需要三个条件：1.自反性，即 a 等于 a；2.对称性，即如果 a 大于 b，则 b 小于 a；3.传递性，即如果 a 小于 b，b 小于 c，则 a 小于 c。现在分别予以证明。

1.

自反性。显然有 aa=aa，所以 a=a。

2. 对称性。如果 a 小于 b，则 ab<ba，所以 ba>ab。因此 b 大于 a。

3. 传递性。如果 a 小于 b，则 ab<ba。当 a 和 b 用十进制表示的时候分别为 l 位和 m 位时，$ab=a\times10^m+b$，$ba=b\times10^l+a$。所以 $a\times10^m+b<b\times10^l+a$。于是有 $a\times10^m-a<b\times10^l-b$，即

$a(10^m-1)<b(10^l-1)$。所以 $a/(10^l-1)<b/(10^m-1)$。

如果 b 小于 c，则 bc<cb。当 c 表示成十进制时为 m 位。和前面证明过程一样，可以得到 $b/(10^m-1)<c/(10^n-1)$。

所以 $a/(10^l-1)< c/(10^n-1)$。于是 $a(10^n-1)<c(10^l-1)$，所以 $a\times10^n+c<c\times10^l+a$，即 ac<ca。所以 a 小于 c。

在证明了我们排序规则的有效性之后，我们接着证明算法的正确性。我们用反证法来证明。

我们把 n 个数按照前面的排序规则排好顺序之后，表示为 $A_1A_2A_3\cdots A_n$。我们假设这样排出来的两个数并不是最小的。即至少存在两个 x 和 y（0<x<y<n），交换第 x 个数和地 y 个数后，$A_1A_2\cdots A_y\cdots A_x\cdots A_n<A_1A_2\cdots A_x\cdots A_y\cdots A_n$。

由于 $A_1A_2\cdots A_x\cdots A_y\cdots A_n$ 是按照前面的规则排好的序列，所以有 $A_x<A_{x+1}<A_{x+2}<\cdots<A_{y-2}<A_{y-1}<A_y$。

由于 A_{y-1} 小于 A_y，所以 $A_{y-1}A_y<A_yA_{y-1}$。我们在序列 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n$ 交换 A_{y-1} 和 A_y，有 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n<A_1A_2\cdots A_x\cdots A_yA_{y-1}\cdots A_n$（这个实际上也需要证明。感兴趣的读者可以自己试着证明）。我们就这样一直把 A_y 和前面的数字交换，直到和 A_x 交换为止。于是就有 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n<A_1A_2\cdots A_x\cdots A_yA_{y-1}\cdots A_n<A_1A_2\cdots A_x\cdots A_yA_{y-2}A_{y-1}\cdots A_n<\cdots<A_1A_2\cdots A_yA_x\cdots A_{y-2}A_{y-1}\cdots A_n$。

同理由于 A_x 小于 A_{x+1}，所以 $A_xA_{x+1}<A_{x+1}A_x$。我们在序列 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n$ 仅仅只交换 A_x 和 A_{x+1}，有 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n<A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n$。我们接下来一直拿 A_x 和它后面的数字交换，直到和 A_{y-1} 交换为止。于是就有 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n<A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n<\cdots<A_1A_2\cdots A_yA_{x+1}A_{x+2}\cdots A_{y-2}A_{y-1}A_x\cdots A_n$。

所以 $A_1A_2\cdots A_x\cdots A_y\cdots A_n< A_1A_2\cdots A_y\cdots A_x\cdots A_n$。这和我们的假设的 $A_1A_2\cdots A_y\cdots A_x\cdots A_n<A_1A_2\cdots A_x\cdots A_y\cdots A_n$ 相矛盾。

所以假设不成立，我们的算法是正确的。

(42)-旋转数组的最小元素

题目：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个排好序的数组的一个旋转，输出旋转数组的最小元素。例如数组{3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为 1。

分析：这道题最直观的解法并不难。从头到尾遍历数组一次，就能找出最小的元素，时间复杂度显然是 O(*N*)。但这个思路没有利用输入数组的特性，我们应该能找到更好的解法。

我们注意到旋转之后的数组实际上可以划分为两个排序的子数组，而且前面的子数组的元素都大于或者等于后面子数组的元素。我们还可以注意到最小的元素刚好是这两个子数组的分界线。我们试着用二元查找法的思路在寻找这个最小的元素。

首先我们用两个指针，分别指向数组的第一个元素和最后一个元素。按照题目旋转的规则，第一个元素应该是大于或者等于最后一个元素的（这其实不完全对，还有特例。后面再讨论特例）。

接着我们得到处在数组中间的元素。如果该中间元素位于前面的递增子数组，那么它应该大

于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一指针指向该中间元素，这样可以缩小寻找的范围。同样，如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。我们可以把第二个指针指向该中间元素，这样同样可以缩小寻找的范围。我们接着再用更新之后的两个指针，去得到和比较新的中间元素，循环下去。按照上述的思路，我们的第一个指针总是指向前面递增数组的元素，而第二个指针总是指向后面递增数组的元素。最后第一个指针将指向前面子数组的最后一个元素，而第二个指针会指向后面子数组的第一个元素。也就是它们最终会指向两个相邻的元素，而第二个指针指向的刚好是最小的元素。这就是循环结束的条件。

基于这个思路，我们可以写出如下代码：

```
// Get the minimum number of a roatation of a sorted array
int Min(int *numbers, int length)
{
    if(numbers == 0 || length <= 0)
        throw new std::exception("Invalid parameters");

    int index1 = 0;
    int index2 = length - 1;
    int indexMid = index1;
    while(numbers[index1] >= numbers[index2])
    {
        if(index2 - index1 == 1)
        {
            indexMid = index2;
            break;
        }

        indexMid = (index1 + index2) / 2;
        if(numbers[indexMid] >= numbers[index1])
            index1 = indexMid;
        else if(numbers[indexMid] <= numbers[index2])
            index2 = indexMid;
    }

    return numbers[indexMid];
}
```

由于我们每次都把寻找的范围缩小一半，该算法的时间复杂度是 O(log*N*)

值得注意的是，如果在面试现场写代码，通常我们需要用一些测试用例来验证代码是不是正确的，我们在验证的时候尽量能考虑全面些。像这道题，我们出来最简单测试用例之外，我们至少还需要考虑如下的情况：

1. 把数组前面的 0 个元素从最前面搬到最后面，也就是原数组不做改动，根据题目的规则这也是一个旋转，此时数组的第一个元素是大于小于或者等于最后一个元素的；
2. 排好序的数组中有可能有相等的元素，我们特别需要注意两种情况。一是旋转之后的数组中，第一个元素和最后一个元素是相等的；另外一种情况是最小的元素在数组中重

复出现

3. 在前面的代码中，如果输入的数组不是一个排序数组的旋转，那将陷入死循环。因此我们需要跟面试官讨论是不是需要判断数组的有效性。在面试的时候，面试官讨论如何验证输入的有效性，能显示我们思维的严密性。本文假设在调用函数 Min 之前，已经验证过输入的有效性了。

最后需要指出的是，如果输入的数组指针是非法指针，我们是用异常来做错误处理。这是因为在这种情况下，如果我们用 return 来结束该函数，返回任何数字都不是正确的。关于无效输入时的函数如何返回错误信息并结束，本博客第 17 题有更详细的讨论，可以参考。

(43)-n 个骰子的点数

题目：把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 S。输入 n，打印出 S 的所有可能的值出现的概率。

分析：玩过麻将的都知道，骰子一共 6 个面，每个面上都有一个点数，对应的数字是 1 到 6 之间的一个数字。所以，n 个骰子的点数和的最小值为 n，最大值为 6n。因此，一个直观的思路就是定义一个长度为 6n-n 的数组，和为 S 的点数出现的次数保存到数组第 S-n 个元素里。另外，我们还知道 n 个骰子的所有点数的排列数 6^n。一旦我们统计出每一点数出现的次数之后，因此只要把每一点数出现的次数除以 n^6，就得到了对应的概率。

该思路的关键就是统计每一点数出现的次数。要求出 n 个骰子的点数和，我们可以先把 n 个骰子分为两堆：第一堆只有一个，另一个有 n-1 个。单独的那一个有可能出现从 1 到 6 的点数。我们需要计算从 1 到 6 的每一种点数和剩下的 n-1 个骰子来计算点数和。接下来把剩下的 n-1 个骰子还是分成两堆，第一堆只有一个，第二堆有 n-2 个。我们把上一轮那个单独骰子的点数和这一轮单独骰子的点数相加，再和剩下的 n-2 个骰子来计算点数和。分析到这里，我们不难发现，这是一种递归的思路。递归结束的条件就是最后只剩下一个骰子了。

基于这种思路，我们可以写出如下代码：

```
int g_maxValue = 6;

void PrintSumProbabilityOfDices_1(int number)
{
    if(number < 1)
        return;

    int maxSum = number * g_maxValue;
    int* pProbabilities = new int[maxSum - number + 1];
    for(int i = number; i <= maxSum; ++i)
        pProbabilities[i - number] = 0;

    SumProbabilityOfDices(number, pProbabilities);

    int total = pow((float)g_maxValue, number);
    for(int i = number; i <= maxSum; ++i)
    {
```

```
            float ratio = (float)pProbabilities[i - number] / total;
            printf("%d: %f\n", i, ratio);
        }

        delete[] pProbabilities;
}

void SumProbabilityOfDices(int number, int* pProbabilities)
{
        for(int i = 1; i <= g_maxValue; ++i)
            SumProbabilityOfDices(number, number, i, 0, pProbabilities);
}

void SumProbabilityOfDices(int original, int current, int value, int tempSum, int* pProbabilities)
{
        if(current == 1)
        {
            int sum = value + tempSum;
            pProbabilities[sum - original]++;
        }
        else
        {
            for(int i = 1; i <= g_maxValue; ++i)
            {
                int sum = value + tempSum;
                SumProbabilityOfDices(original, current - 1, i, sum, pProbabilities);
            }
        }
}
```

上述算法当 number 比较小的时候表现很优异。但由于该算法基于递归，它有很多计算是重复的，从而导致当 number 变大时性能让人不能接受。关于递归算法的性能讨论，详见本博客系列的第 16 题。

我们可以考虑换一种思路来解决这个问题。我们可以考虑用两个数组来存储骰子点数每一总数出现的次数。在一次循环中，第一个数组中的第 n 个数字表示骰子和为 n 出现的次数。那么在下一循环中，我们加上一个新的骰子。那么此时和为 n 的骰子出现的次数，应该等于上一次循环中骰子点数和为 n-1、n-2、n-3、n-4、n-5 与 n-6 的总和。所以我们把另一个数组的第 n 个数字设为前一个数组对应的第 n-1、n-2、n-3、n-4、n-5 与 n-6 之和。基于这个思路，我们可以写出如下代码：

```
void PrintSumProbabilityOfDices_2(int number)
{
        double* pProbabilities[2];
        pProbabilities[0] = new double[g_maxValue * number + 1];
        pProbabilities[1] = new double[g_maxValue * number + 1];
        for(int i = 0; i < g_maxValue * number + 1; ++i)
```

```
    {
        pProbabilities[0][i] = 0;
        pProbabilities[1][i] = 0;
    }

    int flag = 0;
    for (int i = 1; i <= g_maxValue; ++i)
        pProbabilities[flag][i] = 1;

    for (int k = 2; k <= number; ++k)
    {
        for (int i = k; i <= g_maxValue * k; ++i)
        {
            pProbabilities[1 - flag][i] = 0;
       for(int j = 1; j <= i && j <= g_maxValue; ++j)
                pProbabilities[1 - flag][i] += pProbabilities[flag][i - j];
        }

        flag = 1 - flag;
    }

    double total = pow((double)g_maxValue, number);
    for(int i = number; i <= g_maxValue * number; ++i)
    {
        double ratio = pProbabilities[flag][i] / total;
        printf("%d: %f\n", i, ratio);
    }

    delete[] pProbabilities[0];
    delete[] pProbabilities[1];
}
```

值得提出来的是，上述代码没有在函数里把一个骰子的最大点数硬编码(hard code)为 6，而是用一个变量 g_maxValue 来表示。这样做的好处时，如果某个厂家生产了最大点数为 4 或者 8 的骰子，我们只需要在代码中修改一个地方，扩展起来很方便。如果在面试的时候我们能对面试官提起对程序扩展性的考虑，一定能给面试官留下一个很好的印象。

(44)一数值的整数次方

题目：实现函数 double Power(double base, int exponent)，求 base 的 exponent 次方。不需要考虑溢出。

分析：这是一道看起来很简单的问题。可能有不少的人在看到题目后 30 秒写出如下的代码：

```
double Power(double base, int exponent)
```

```
{
    double result = 1.0;
    for(int i = 1; i <= exponent; ++i)
        result *= base;

    return result;
}
```

上述代码至少有一个问题：由于输入的 exponent 是个 int 型的数值，因此可能为正数，也可能是负数。上述代码只考虑了 exponent 为正数的情况。
接下来，我们把代码改成：

```
bool g_InvalidInput = false;

double Power(double base, int exponent)
{
    g_InvalidInput = false;

    if(IsZero(base) && exponent < 0)
    {
        g_InvalidInput = true;
        return 0.0;
    }

    unsigned int unsignedExponent = static_cast<unsigned int>(exponent);
    if(exponent < 0)
        unsignedExponent = static_cast<unsigned int>(-exponent);

    double result = PowerWithUnsignedExponent(base, unsignedExponent);
    if(exponent < 0)
        result = 1.0 / result;

    return result;
}

double PowerWithUnsignedExponent(double base, unsigned int exponent)
{
    double result = 1.0;
    for(int i = 1; i <= exponent; ++i)
        result *= base;

    return result;
}
```

上述代码较之前的代码有了明显的改进，主要体现在：首先它考虑了 exponent 为负数的情

况。其次它还特殊处理了当底数 base 为 0 而指数 exponent 为负数的情况。如果没有特殊处理，就有可能出现除数为 0 的情况。这里是用一个全局变量来表示无效输入。关于不同方法来表示输入无效的讨论，详见本系列第 17 题。
最后需要指出的是：由于 0^0 次方在数学上是没有意义的，因此无论是输入 0 还是 1 都是可以接受的，但需要在文档中说明清楚。
这次的代码在逻辑上看起来已经是很严密了，那是不是意味了就没有进一步改进的空间了呢？接下来我们来讨论一下它的性能：
如果我们输入的指数 exponent 为 32，按照前面的算法，我们在函数 PowerWithUnsignedExponent 中的循环中至少需要做乘法 31 次。但我们可以换一种思路考虑：我们要求出一个数字的 32 次方，如果我们已经知道了它的 16 次方，那么只要在 16 次方的基础上再平方一次就可以了。而 16 次方是 8 次方的平方。这样以此类推，我们求 32 次方只需要做 5 次乘法：求平方，在平方的基础上求 4 次方，在 4 次方的基础上平方求 8 次方，在 8 次方的基础上求 16 次方，最后在 16 次方的基础上求 32 次方。
32 刚好是 2 的整数次方。如果不巧输入的指数 exponent 不是 2 的整数次方，我们又该怎么办呢？我们换个数字 6 来分析，6 就不是 2 的整数次方。但我们注意到 6 是等于 2+4，因此我们可以把一个数的 6 次方表示为该数的平方乘以它的 4 次方。于是，求一个数的 6 次方需要 3 次乘法：第一次求平方，第二次在平方的基础上求 4 次方，最后一次把平方的结果和 4 次方的结果相乘。
现在把上面的思路总结一下：把指数分解了一个或若干个 2 的整数次方。我们可以用连续平方的方法得到以 2 的整数次方为指数的值，接下来再把每个前面得到的值相乘就得到了最后的结果。
到目前为止，我们还剩下一个问题：如何将指数分解为一个或若干个 2 的整数次方。我们把指数表示为二进制数再来分析。比如 6 的二进制表示为 110，在它的第 2 位和第 3 位为 1，因此 6=2^(2-1)+2^(3-1) 。也就是说只要它的第 n 位为 1，我们就加上 2 的 n-1 次方。
最后，我们根据上面的思路，重写函数 PowerWithUnsignedExponent：

```
double PowerWithUnsignedExponent(double base, unsigned int exponent)
{
    std::bitset<32> bits(exponent);
    if(bits.none())
        return 1.0;

    int numberOf1 = bits.count();
    double multiplication[32];
    for(int i = 0; i < 32; ++i)
    {
        multiplication[i] = 1.0;
    }

    // if the i-th bit in exponent is 1,
    // the i-th number in array multiplication is base ^ (2 ^ n)
    int count = 0;
    double power = 1.0;
    for(int i = 0; i < 32 && count < numberOf1; ++i)
```

```
    {
        if(i == 0)
            power = base;
        else
            power = power * power;

        if(bits.at(i))
        {
            multiplication[i] = power;
            ++count;
        }
    }

    power = 1.0;
    for(int i = 0; i < 32; ++i)
    {
        if(bits.at(i))
            power *= multiplication[i];
    }

    return power;
}
```

在上述代码中，我们用 C++的标准函数库中 bitset 把整数表示为它的二进制，增大代码的可读性。如果 exponent 的第 i 位为 1，那么在数组 multiplication 的第 i 个数字中保存以 base 为底数，以 2 的 i 次方为指数的值。最后，我们再把所以位为 1 在数组中的对应的值相乘得到最后的结果。

上面的代码需要我们根据 base 的二进制表达的每一位来确定是不是需要做乘法。对二进制的操作很多人都不是很熟悉，因此编码可能觉得有些难度。我们可以换一种思路考虑：我们要求出一个数字的 32 次方，如果我们已经知道了它的 16 次方，那么只要在 16 次方的基础上再平方一次就可以了。而 16 次方是 8 次方的平方。这样以此类推，我们求 32 次方只需要做 5 次乘法：先求平方，在平方的基础上求 4 次方，在 4 次方的基础上平方求 8 次方，在 8 次方的基础上求 16 次方，最后在 16 次方的基础上求 32 次方。
也就是说，我们可以用如下公式求 a 的 n 次方：

$$a^n = \begin{cases} a^{n/2} * a^{n/2} & n\text{为偶数} \\ a^{(n-1)/2} * a^{(n-1)/2} * a & n\text{为奇数} \end{cases}$$

这个公式很容易就能用递归来实现。新的 PowerWithUnsignedExponent 代码如下：

```
double PowerWithUnsignedExponent(double base, unsigned int exponent)
{
    if(exponent == 0)
        return 1;
```

```
    if(exponent == 1)
        return base;

    double result = PowerWithUnsignedExponent(base, exponent >> 1);
    result *= result;
    if(exponent & 0x1 == 1)
        result *= base;

    return result;
}
```

(45)－Singleton

题目：设计一个类，我们只能生成该类的一个实例。
分析：只能生成一个实例的类是实现了 Singleton 模式的类型。
由于设计模式在面向对象程序设计中起着举足轻重的作用，在面试过程中很多公司都喜欢问一些与设计模式相关的问题。在常用的模式中，Singleton 是唯一一个能够用短短几十行代码完整实现的模式。因此，写一个 Singleton 的类型是一个很常见的面试题。
事实上，要让一个类型是能创建一个实例不是一件很难的事情。我们可以把该类型的构造函数设为 private，这样该类型的用户就不能创建该类型的实例了。然后我们在给类型中创建一个静态实例。当用户需要该类型的实例时，我们就返回这个实例。基于这个思路，我们可以用 C#写出如下代码：

```
// We can only get an instance of the class Singleton1. The instance
// is created when class Singleton1 is referenced at the first time
public sealed class Singleton1
{
    private Singleton1()
    {
    }

    private static Singleton1 instance = new Singleton1();
    public static Singleton1 Instance
    {
        get
        {
            return instance;
        }
    }
}
```

由于类 Singleton1 的实例是一个静态变量，因此它会在该类型的第一次引用的时候被创建，而不是第一次在调用 Singleton1.get_Instance 的时候被创建。如果我们此时并不需要该实例，

那么我们就过早地初始化该实例，无论在内存空间还是 CPU 时间上都是一种浪费。
我们可以把上面的代码稍作改动，就能实现在第一次调用 Singleton_getInstance 时候才会创建类型的唯一实例：

```
// We can only get an instance of the class Singleton2.
// The instance is created when we need it explicitly.
public sealed class Singleton2
{
    private Singleton2()
    {
    }

    private static Singleton2 instance = null;
    public static Singleton2 Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton2();

            return instance;
        }
    }
}
```

我们在单线程环境下只能得到类型 Singleton2 的一个实例，但在多线程环境下情况就可能不同了。设想如果两个线程同时运行到语句 if (instance == null)，而此时该实例的确没有创建，那么两个线程都会创建一个实例。此时，类型 Singleton2 就不在满足模式 Singleton 的要求了。
为了保证在多线程环境下我们还是只能得到类型的一个实例，我们应该在判断实例是否已经创建，以及在实例还没有创建的时候创建一个实例的语句上加一个同步锁。我们把 Singleton2 稍作修改就得到了如下代码：

```
// We can only get an instance of the class Singleton3,
// even when there are multiple threads which are trying
// to get an instance concurrently.
public sealed class Singleton3
{
    private Singleton3()
    {
    }

    private static readonly object syncObj = new object();

    private static Singleton3 instance = null;
    public static Singleton3 Instance
```

```
        {
            get
            {
                lock (syncObj)
                {
                    if (instance == null)
                        instance = new Singleton3();
                }

                return instance;
            }
        }
    }
```

说明一下，由于 C/C++没有为线程同步提供直接的支持。为了让代码显得简洁，而不是让大量的代码在实现同步锁而偏离了实现 Singleton 的主题，本文的代码用 C#实现。
我们还是假设有两个线程同时想创建一个实例。由于在一个时刻只能有一个线程能得到同步锁。当第一个线程加上锁时，第二个线程只能在等待。当第一个线程发现实例还没有创建时，它创建出一个实例。接着第一个线程释放同步锁。此时第二个线程可以加上同步锁，并运行接下来的代码。由于此时实例已经被第一个线程创建出来了，第二个线程就不会重复创建实例了。于是保证了我们只能得到一个实例。
但是类型 Singleton3 还不是完美。由于我们每次调用 Singleton3.get_Instance 的时候，都会试图加上一个同步锁。由于加锁是一个非常耗时的操作，在没有必要的时候我们应该尽量避免这样的操作。
实际上，我们只是在实例还没有创建之前需要加锁操作，以保证只有一个线程创建出实例。而当实例已经创建之后，我们已经不需要再做加锁操作了。于是，我们可以把上述代码再作进一步的改进：

```
    // We can only get an instance of the class Singleton4,
    // even when there are multiple threads which are trying
    // to get an instance concurrently. When the instance has
    // been created, we don't need the lock any more.
    public sealed class Singleton4
    {
        private Singleton4()
        {
        }

        private static object syncObj = new object();

        private static Singleton4 instance = null;
        public static Singleton4 Instance
        {
            get
            {
```

```
            if (instance == null)
            {
                lock (syncObj)
                {
                    if (instance == null)
                        instance = new Singleton4();
                }
            }

            return instance;
        }
    }
}
```

我们只需要在最开始调用 Singleton4_getInstance（可能来自一个线程，也可能来自多个线程）的时候需要加锁。当实例已经创建之后，我们就不再需要作加锁操作，从而在后续调用 Singleton4_getInstance 时性能得到提升。

(46)一对称子字符串的最大长度

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。

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。

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（n^3）。
通常 O（n^3）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。
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（n^2）。

(47)-数组中出现次数超过一半的数字

题目：数组中有一个数字出现的次数超过了数组长度的一半，找出这个数字。

分析：这是一道广为流传的面试题，包括百度、微软和 Google 在内的多家公司都曾经采用过这个题目。要几十分钟的时间里很好地解答这道题，除了较好的编程能力之外，还需要较快的反应和较强的逻辑思维能力。

看到这道题，我们马上就会想到，要是这个数组是排序的数组就好了。如果是排序的数组，那么我们只要遍历一次就可以统计出每个数字出现的次数，这样也就能找出符合要求的数字了。题目给出的数组没有说是排好序的，因此我们需要给它排序。排序的时间复杂度是 O(n*log*n)，再加上遍历的时间复杂度 O(n)，因此总的复杂度是 O(n*log*n)。

接下来我们试着看看能不能想出更快的算法。前面思路的时间主要是花在排序上。我们可以创建一个哈希表来消除排序的时间。哈希表的键值（Key）为数组中的数字，值（Value）为该数字对应的次数。有了这个辅助的哈希表之后，我们只需要遍历数组中的每个数字，找到它在哈希表中对应的位置并增加它出现的次数。这种哈希表的方法在数组的所有数字都在一个比较窄的范围内的时候很有效。本博客系列的第 13 题就是一个应用哈希表的例子。不过本题并没有限制数组里数字的范围，我们要么需要创建一个很大的哈希表，要么需要设计一个很复杂的方法来计算哈希值。因此总体说来这个方法还不是很好。

前面两种思路都没有考虑到题目中数组的特性：数组中有个数字出现的次数超过了数组长度的一半。也就是说，有个数字出现的次数比其他所有数字出现次数的和还要多。因此我们可以考虑在遍历数组的时候保存两个值：一个是数组中的一个数字，一个是次数。当我们遍历到下一个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加 1。如果下一个数字和我们之前保存的数字不同，则次数减 1。如果次数为零，我们需要保存下一个数字，并把次数设为 1。由于我们要找的数字出现的次数比其他所有数字出现的次数之和还要多，那么要找的数字肯定是最后一次把次数设为 1 时对应的数字。
基于这个思路，我们不难写出如下代码：

```
bool g_bInputInvalid = false;

///////////////////////////////////////////////////////////////////////
// Input: an array with "length" numbers. A number in the array
// appear more than "length / 2 + 1" times.
// Output: If the input is valid, return the number appearing more than
// "length / 2 + 1" times. Otherwise, return 0 and set flag g_bInputInvalid
// to be true.
///////////////////////////////////////////////////////////////////////
int MoreThanHalfNum(int* numbers, unsigned int length)
{
    if(numbers == NULL && length == 0)
    {
        g_bInputInvalid = true;
        return 0;
    }

    g_bInputInvalid = false;

    int result = numbers[0];
    int times = 1;
    for(int i = 1; i < length; ++i)
    {
        if(times == 0)
        {
            result = numbers[i];
            times = 1;
        }
        else if(numbers[i] == result)
            times++;
        else
            times--;
    }

    // verify whether the input is valid
    times = 0;
```

```
    for(int i = 0; i < length; ++i)
    {
        if(numbers[i] == result)
            times++;
    }

    if(times * 2 <= length)
    {
        g_bInputInvalid = true;
        result = 0;
    }

    return result;
}
```

在上述代码中，有两点值得讨论：

（1） 我们需要考虑当输入的数组或者长度无效时，如何告诉函数的调用者输入无效。关于处理无效输入的几种常用方法，在本博客系列的第 17 题中有详细的讨论；

（2） 本算法的前提是输入的数组中的确包含一个出现次数超过数组长度一半的数字。如果数组中并不包含这么一个数字，那么输入也是无效的。因此在函数结束前我还加了一段代码来验证输入是不是有效的。

(48)-二叉树两个结点的最低共同父结点

题目：二叉树的结点定义如下：

```
struct TreeNode
{
    int m_nvalue;
    TreeNode* m_pLeft;
    TreeNode* m_pRight;
};
```

输入二叉树中的两个结点，输出这两个结点在数中最低的共同父结点。

分析：求数中两个结点的最低共同结点是面试中经常出现的一个问题。这个问题至少有两个变种。

第一变种是二叉树是一种特殊的二叉树：查找二叉树。也就是树是排序过的，位于左子树上的结点都比父结点小，而位于右子树的结点都比父结点大。我们只需要从根结点开始和两个结点进行比较。如果当前结点的值比两个结点都大，则最低的共同父结点一定在当前结点的左子树中。如果当前结点的值比两个结点都小，则最低的共同父结点一定在当前结点的右子树中。

第二个变种是树不一定是二叉树，每个结点都有一个指针指向它的父结点。于是我们可以从任何一个结点出发，得到一个到达树根结点的单向链表。因此这个问题转换为两个单向链表的第一个公共结点。我们在本面试题系列的第 35 题讨论了这个问题。

现在我们回到这个问题本身。所谓共同的父结点，就是两个结点都出现在这个结点的子树中。因此我们可以定义一函数，来判断一个结点的子树中是不是包含了另外一个结点。这不是件很难的事，我们可以用递归的方法来实现：

```
////////////////////////////////////////////////////////////////////////////////
// If the tree with head pHead has a node pNode, return true.
// Otherwise return false.
////////////////////////////////////////////////////////////////////////////////
bool HasNode(TreeNode* pHead, TreeNode* pNode)
{
    if(pHead == pNode)
        return true;

    bool has = false;

    if(pHead->m_pLeft != NULL)
        has = HasNode(pHead->m_pLeft, pNode);

    if(!has && pHead->m_pRight != NULL)
        has = HasNode(pHead->m_pRight, pNode);

    return has;
}
```

我们可以从根结点开始，判断以当前结点为根的树中左右子树是不是包含我们要找的两个结点。如果两个结点都出现在它的左子树中，那最低的共同父结点也出现在它的左子树中。如果两个结点都出现在它的右子树中，那最低的共同父结点也出现在它的右子树中。如果两个结点一个出现在左子树中，一个出现在右子树中，那当前的结点就是最低的共同父结点。基于这个思路，我们可以写出如下代码：

```
////////////////////////////////////////////////////////////////////////////////
// Find the last parent of pNode1 and pNode2 in a tree with head pHead
////////////////////////////////////////////////////////////////////////////////
TreeNode* LastCommonParent_1(TreeNode* pHead, TreeNode* pNode1, TreeNode* pNode2)
{
    if(pHead == NULL || pNode1 == NULL || pNode2 == NULL)
        return NULL;

    // check whether left child has pNode1 and pNode2
    bool leftHasNode1 = false;
    bool leftHasNode2 = false;
    if(pHead->m_pLeft != NULL)
    {
        leftHasNode1 = HasNode(pHead->m_pLeft, pNode1);
        leftHasNode2 = HasNode(pHead->m_pLeft, pNode2);
    }
```

```
    if(leftHasNode1 && leftHasNode2)
    {
        if(pHead->m_pLeft == pNode1 || pHead->m_pLeft == pNode2)
            return pHead;

        return LastCommonParent_1(pHead->m_pLeft, pNode1, pNode2);
    }

    // check whether right child has pNode1 and pNode2
    bool rightHasNode1 = false;
    bool rightHasNode2 = false;
    if(pHead->m_pRight != NULL)
    {
        if(!leftHasNode1)
            rightHasNode1 = HasNode(pHead->m_pRight, pNode1);
        if(!leftHasNode2)
            rightHasNode2 = HasNode(pHead->m_pRight, pNode2);
    }

    if(rightHasNode1 && rightHasNode2)
    {
        if(pHead->m_pRight == pNode1 || pHead->m_pRight == pNode2)
            return pHead;

        return LastCommonParent_1(pHead->m_pRight, pNode1, pNode2);
    }

    if((leftHasNode1 && rightHasNode2)
        || (leftHasNode2 && rightHasNode1))
        return pHead;

    return NULL;
}
```

接着我们来分析一下这个方法的效率。函数 HasNode 的本质就是遍历一棵树，其时间复杂度是 O(n)（n 是树中结点的数目）。由于我们根结点开始，要对每个结点调用函数 HasNode。因此总的时间复杂度是 $O(n^2)$。

我们仔细分析上述代码，不难发现我们判断以一个结点为根的树是否含有某个结点时，需要遍历树的每个结点。接下来我们判断左子结点或者右结点为根的树中是否含有要找结点，仍然需要遍历。第二次遍历的操作其实在前面的第一次遍历都做过了。由于存在重复的遍历，本方法在时间效率上肯定不是最好的。

前面我们提过如果结点中有一个指向父结点的指针，我们可以把问题转化为求两个链表的共同结点。现在我们可以想办法得到这个链表。我们在本面试题系列的第 4 题中分析过如何得到一条中根结点开始的路径。我们在这里稍作变化即可：

//

```
// Get the path form pHead and pNode in a tree with head pHead
////////////////////////////////////////////////////////////////////////////////
bool GetNodePath(TreeNode* pHead, TreeNode* pNode, std::list<TreeNode*>& path)
{
    if(pHead == pNode)
        return true;

    path.push_back(pHead);

    bool found = false;
    if(pHead->m_pLeft != NULL)
        found = GetNodePath(pHead->m_pLeft, pNode, path);
    if(!found && pHead->m_pRight)
        found = GetNodePath(pHead->m_pRight, pNode, path);

    if(!found)
        path.pop_back();

  return found;
}
```

由于这个路径是从跟结点开始的。最低的共同父结点就是路径中的最后一个共同结点：

```
////////////////////////////////////////////////////////////////////////////////
// Get the last common Node in two lists: path1 and path2
////////////////////////////////////////////////////////////////////////////////
TreeNode* LastCommonNode
(
    const std::list<TreeNode*>& path1,
    const std::list<TreeNode*>& path2
)
{
    std::list<TreeNode*>::const_iterator iterator1 = path1.begin();
    std::list<TreeNode*>::const_iterator iterator2 = path2.begin();

    TreeNode* pLast = NULL;

    while(iterator1 != path1.end() && iterator2 != path2.end())
    {
        if(*iterator1 == *iterator2)
            pLast = *iterator1;

        iterator1++;
        iterator2++;
    }
```

```
    return pLast;
}
```

有了前面两个子函数之后，求两个结点的最低共同父结点就很容易了。我们先求出从根结点出发到两个结点的两条路径，再求出两条路径的最后一个共同结点。代码如下：

```
////////////////////////////////////////////////////////////////////////////////
// Find the last parent of pNode1 and pNode2 in a tree with head pHead
////////////////////////////////////////////////////////////////////////////////
TreeNode* LastCommonParent_2(TreeNode* pHead, TreeNode* pNode1, TreeNode* pNode2)
{
    if(pHead == NULL || pNode1 == NULL || pNode2 == NULL)
        return NULL;

    std::list<TreeNode*> path1;
    GetNodePath(pHead, pNode1, path1);

    std::list<TreeNode*> path2;
    GetNodePath(pHead, pNode2, path2);

    return LastCommonNode(path1, path2);
}
```

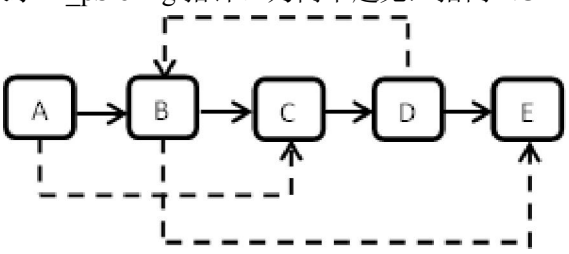
这种思路的时间复杂度是 O(n)，时间效率要比第一种方法好很多。但同时我们也要注意到，这种思路需要两个链表来保存路径，空间效率比不上第一个方法。

(49)-复杂链表的复制

题目：有一个复杂链表，其结点除了有一个 m_pNext 指针指向下一个结点外，还有一个 m_pSibling 指向链表中的任一结点或者 NULL。其结点的 C++定义如下：

```
        struct ComplexNode
{
   int m_nValue;
   ComplexNode* m_pNext;
   ComplexNode* m_pSibling;
};
```

下图是一个含有 5 个结点的该类型复杂链表。图中实线箭头表示 m_pNext 指针，虚线箭头表示 m_pSibling 指针。为简单起见，指向 NULL 的指针没有画出。



请完成函数 ComplexNode* Clone(ComplexNode* pHead)，以复制一个复杂链表。

分析：在常见的数据结构上稍加变化，这是一种很新颖的面试题。要在不到一个小时的时间里解决这种类型的题目，我们需要较快的反应能力，对数据结构透彻的理解以及扎实的编程功底。

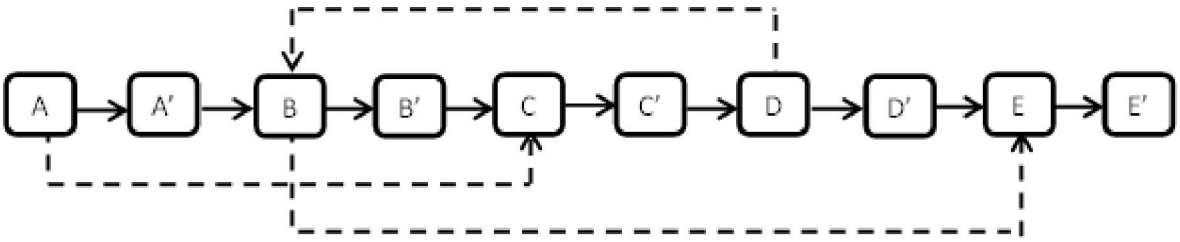
看到这个问题，我的第一反应是分成两步：第一步是复制原始链表上的每个链表，并用 m_pNext 链接起来。第二步，假设原始链表中的某节点 N 的 m_pSibling 指向结点 S。由于 S 的位置在链表上有可能在 N 的前面也可能在 N 的后面，所以要定位 N 的位置我们需要从原始链表的头结点开始找。假设从原始链表的头结点开始经过 s 步找到结点 S。那么在复制链表上结点 N 的 m_pSibling 的 S’，离复制链表的头结点的距离也是 s。用这种办法我们就能为复制链表上的每个结点设置 m_pSibling 了。

对一个含有 n 个结点的链表，由于定位每个结点的 m_pSibling，都需要从链表头结点开始经过 O(n)步才能找到，因此这种方法的总时间复杂度是 $O(n^2)$。

由于上述方法的时间主要花费在定位结点的 m_pSibling 上面，我们试着在这方面去做优化。我们还是分为两步：第一步仍然是复制原始链表上的每个结点 N，并创建 N’，然后把这些创建出来的结点链接起来。这里我们对<N，N’>的配对信息放到一个哈希表中。第二步还是设置复制链表上每个结点的 m_pSibling。如果在原始链表中结点 N 的 m_pSibling 指向结点 S，那么在复制链表中，对应的 N’应该指向 S’。由于有了哈希表，我们可以用 O(1)的时间根据 S 找到 S’。

第二种方法相当于用空间换时间，以 O(n)的空间消耗实现了 O(n)的时间效率。

接着我们来换一种思路，在不用辅助空间的情况下实现 O(n)的时间效率。第三种方法的第一步仍然是根据原始链表的每个结点 N，创建对应的 N’。这一次，我们把 N’链接在 N 的后面。实例中的链表经过这一步之后变成了：

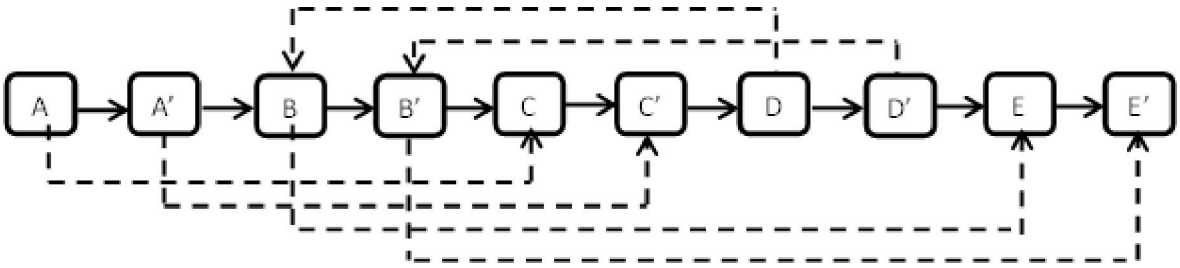


这一步的代码如下：

```
//////////////////////////////////////////////////////////////////////////////////
// Clone all nodes in a complex linked list with head pHead,
// and connect all nodes with m_pNext link
//////////////////////////////////////////////////////////////////////////////////
void CloneNodes(ComplexNode* pHead)
{
    ComplexNode* pNode = pHead;
    while(pNode != NULL)
    {
        ComplexNode* pCloned = new ComplexNode();
        pCloned->m_nValue = pNode->m_nValue;
        pCloned->m_pNext = pNode->m_pNext;
        pCloned->m_pSibling = NULL;
```

```
        pNode->m_pNext = pCloned;

        pNode = pCloned->m_pNext;
    }
}
```

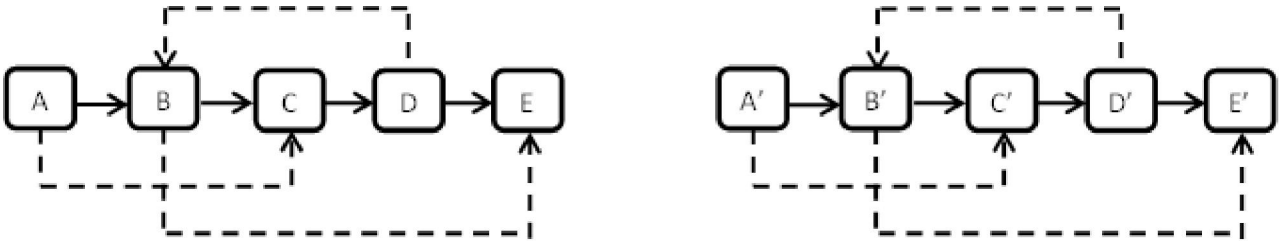
第二步是设置我们复制出来的链表上的结点的 m_pSibling。假设原始链表上的 N 的 m_pSibling 指向结点 S，那么其对应复制出来的 N’是 N->m_pNext，同样 S’也是 S->m_pNext。这就是我们在上一步中把每个结点复制出来的结点链接在原始结点后面的原因。有了这样的链接方式，我们就能在 O(1)中就能找到每个结点的 m_pSibling 了。例子中的链表经过这一步，就变成如下结构了：



这一步的代码如下：

```
//////////////////////////////////////////////////////////////////////////////
// Connect sibling nodes in a complex link list
//////////////////////////////////////////////////////////////////////////////
void ConnectSiblingNodes(ComplexNode* pHead)
{
    ComplexNode* pNode = pHead;
    while(pNode != NULL)
    {
        ComplexNode* pCloned = pNode->m_pNext;
        if(pNode->m_pSibling != NULL)
        {
            pCloned->m_pSibling = pNode->m_pSibling->m_pNext;
        }

        pNode = pCloned->m_pNext;
    }
}
```

第三步是把这个长链表拆分成两个：把奇数位置的结点链接起来就是原始链表，把偶数位置的结点链接出来就是复制出来的链表。上述例子中的链表拆分之后的两个链表如下：



要实现这一步，也不是很难的事情。其对应的代码如下：

```
//////////////////////////////////////////////////////////////////////////////
// Split a complex list into two:
// Reconnect nodes to get the original list, and its cloned list
//////////////////////////////////////////////////////////////////////////////
ComplexNode* ReconnectNodes(ComplexNode* pHead)
{
    ComplexNode* pNode = pHead;
    ComplexNode* pClonedHead = NULL;
    ComplexNode* pClonedNode = NULL;

    if(pNode != NULL)
    {
        pClonedHead = pClonedNode = pNode->m_pNext;
        pNode->m_pNext = pClonedNode->m_pNext;
        pNode = pNode->m_pNext;
    }

    while(pNode != NULL)
    {
        pClonedNode->m_pNext = pNode->m_pNext;
        pClonedNode = pClonedNode->m_pNext;

        pNode->m_pNext = pClonedNode->m_pNext;
        pNode = pNode->m_pNext;
    }

    return pClonedHead;
}
```

我们把上面三步合起来，就是复制链表的完整过程：

```
//////////////////////////////////////////////////////////////////////////////
// Clone a complex linked list with head pHead
//////////////////////////////////////////////////////////////////////////////
ComplexNode* Clone(ComplexNode* pHead)
{
    CloneNodes(pHead);
    ConnectSiblingNodes(pHead);
    return ReconnectNodes(pHead);
```

}

(50)-树为另一树的子结构

题目：二叉树的结点定义如下：

```
struct TreeNode
{
    int m_nValue;
    TreeNode* m_pLeft;
    TreeNode* m_pRight;
};
```

输入两棵二叉树 A 和 B，判断树 B 是不是 A 的子结构。

例如，下图中的两棵树 A 和 B，由于 A 中有一部分子树的结构和 B 是一样的，因此 B 就是 A 的子结构。

分析：这是 2010 年微软校园招聘时的一道题目。二叉树一直是微软面试题中经常出现的数据结构。对微软有兴趣的读者一定要重点关注二叉树。

回到这个题目的本身。要查找树 A 中是否存在和树 B 结构一样的子树，我们可以分为两步：第一步在树 A 中找到和 B 的根结点的值一样的结点 N，第二步再判断树 A 中以 N 为根结点的子树是不是包括和树 B 一样的结构。

第一步在树 A 中查找与根结点的值一样的结点。这实际上就是树的遍历。对二叉树这种数据结构熟悉的读者自然知道我们可以用递归的方法去遍历，也可以用循环的方法去遍历。由于递归的代码实现比较简洁，面试时如果没有特别要求，我们通常都会采用递归的方式。下面是参考代码：

```
bool HasSubtree(TreeNode* pTreeHead1, TreeNode* pTreeHead2)
{
    if((pTreeHead1 == NULL && pTreeHead2 != NULL) ||
        (pTreeHead1 != NULL && pTreeHead2 == NULL))
        return false;

    if(pTreeHead1 == NULL && pTreeHead2 == NULL)
        return true;

    return HasSubtreeCore(pTreeHead1, pTreeHead2);
}
bool HasSubtreeCore(TreeNode* pTreeHead1, TreeNode* pTreeHead2)
```

```
{
    bool result = false;
    if(pTreeHead1->m_nValue == pTreeHead2->m_nValue)
    {
        result = DoesTree1HaveAllNodesOfTree2(pTreeHead1, pTreeHead2);
    }

    if(!result && pTreeHead1->m_pLeft != NULL)
        result = HasSubtreeCore(pTreeHead1->m_pLeft, pTreeHead2);

    if(!result && pTreeHead1->m_pRight != NULL)
        result = HasSubtreeCore(pTreeHead1->m_pRight, pTreeHead2);

    return result;
}
```

在上述代码中，我们递归调用 hasSubtreeCore 遍历二叉树 A。如果发现某一结点的值和树 B 的头结点的值相同，则调用 DoesTree1HaveAllNodeOfTree2，做第二步判断。

在面试的时候，我们一定要注意边界条件的检查，即检查空指针。当树 A 或树 B 为空的时候，定义相应的输出。如果没有检查并做相应的处理，程序非常容易崩溃，这是面试时非常忌讳的事情。由于没有必要在每一次递归中做边界检查（每一次递归都做检查，增加了不必要的时间开销），上述代码只在 HasSubtree 中作了边界检查后，在 HasSubtreeCore 中作递归遍历。

接下来考虑第二步，判断以树 A 中以 N 为根结点的子树是不是和树 B 具有相同的结构。同样，我们也可以用递归的思路来考虑：如果结点 N 的值和树 B 的根结点不相同，则以 N 为根结点的子树和树 B 肯定不具有相同的结点；如果他们的值相同，则递归地判断他们的各自的左右结点的值是不是相同。递归的终止条件是我们到达了树 A 或者树 B 的叶结点。参考代码如下：

```
bool DoesTree1HaveAllNodesOfTree2(TreeNode* pTreeHead1, TreeNode* pTreeHead2)
{
    if(pTreeHead2 == NULL)
        return true;

    if(pTreeHead1 == NULL)
        return false;

    if(pTreeHead1->m_nValue != pTreeHead2->m_nValue)
        return false;

    return DoesTree1HaveAllNodesOfTree2(pTreeHead1->m_pLeft, pTreeHead2->m_pLeft)
&&
        DoesTree1HaveAllNodesOfTree2(pTreeHead1->m_pRight, pTreeHead2->m_pRight);
}
```

(51)-顺时针打印矩阵

题目：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。
例如：如果输入如下矩阵：
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
则依次打印出数字 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10。
分析：第一次看到这个题目的时候，觉得这个题目很简单，完全不需要用到数据结构或者算法的知识，因此没有兴趣做这道题。后来听到包括 Autodesk、EMC 在内的多家公司在面试或者笔试里采用过这道题，于是想这么多家公司用它来检验一个程序员的编程功底总是有原因的，于是决定自己写一遍试一下。真正写一遍才发现，要完整写出这道题的代码，还真不是件容易的事情。

解决这道题的难度在于代码中会包含很多个循环，而且还有多个边界条件需要判断。如果在把问题考虑得很清楚之前就开始写代码，不可避免地会越写越混乱。因此解决这个问题的关键，在于先要形成清晰的思路，并把复杂的问题分解成若干个简单的问题。下面分享我分析这个问题的过程。

通常当我们遇到一个复杂的问题的时候，我们可以用图形帮助我们思考。由于我们是以从外圈到内圈的顺序依次打印，我们在矩阵中标注一圈作为我们分析的目标。在下图中，我们设矩阵的宽度为 columns，而其高度为 rows。我们我们选取左上角坐标为(startX, startY)，右下角坐标为(endX, endY)的一个圈来分析。

由于 endX 和 endY 可以根据 startX、startY 以及 columns、rows 来求得，因此此时我们只需要引入 startX 和 startY 两个变量。我们可以想象有一个循环，在每一次循环里我们从(startX, startY)出发按照顺时针打印数字。
接着我们分析这个循环结束的条件。对一个 5×5 的矩阵而言，最后一圈只有一个数字，对应的坐标为(2, 2)。我们发现 5 > 2 * 2。对一个 6×6 的矩阵而言，最后一圈有四个数字，对应的坐标仍然为(2, 2)。我们发现 6 > 2 * 2 依然成立。于是我们可以得出，让循环继续的条件是 columns > startX * 2 && rows > startY * 2。有了这些分析，我们就可以写出如下的代码：

```
void PrintMatrixClockwisely(int** numbers, int columns, int rows)
{
    if(numbers == NULL || columns <= 0 || rows <= 0)
        return;

    int startX = 0;
    int startY = 0;
```

```
    while(columns > startX * 2 && rows > startY * 2)
    {
        PrintMatrixInCircle(numbers, columns, rows, startX, startY);

        ++startX;
        ++startY;
    }
}
```

接下来我们分析如何在 PrintMatrixInCircle 中按照顺时针的顺序打印一圈的数字。如同在图中标注的那样，我们可以分四步来打印：第一步是从左到右打印一行（上图中黄色区域），第二步是从上到下打印一列（上图中绿色区域），第三步从右到左打印一行（上图中蓝色区域），最后一步是从下到上打印一列（上图中紫色区域）。也就是我们把打印一圈数字这个问题，分解成四个子问题。我们可以为每个子问题定义一个函数。四个步骤对应的函数名称我们分别定义为：PrintARowIncreasingly，PrintAColumnIncreasingly，PrintARowDecreasingly 和 PrintAColumnDecreasingly。

现在我们暂时不考虑如何去实现这四个函数，而是先考虑我们需要分别给这些函数传入哪些参数。第一步打印一行时，所有的数字的行号是固定的（startY），不同数字的列号不同。我们需要传入一个起始列号（startX）和终止列号（endX）。第二步打印一列时，所有的数字的列号是固定的，不同的数字的行号不同。我们需要传入一个起始行号（startY + 1）和一个终止行号（endY）。第三步和第四步和前面两步类似，读者可以自己分析。

接下来我们需要考虑特殊情况。并不是所有数字圈都需要四步来打印。比如当一圈退化成一行的时候，也就是 startY 等于 endY 的时候，我们只需要第一步就把所有的数字都打印完了，其余的步骤都是多余的。因此我们需要考虑第二、三、四步打印的条件。根据前面我们分析，不难发现打印第二步的条件是 startY < endY。对于第三步而言，如果 startX 等于 endX，也就是这一圈中只有一列数字，那么所有的数字都在第二步打印完了；如果 startY 等于 endY，也就是这一圈中只有一行数字，那么所有的数字都在第一步打印完了。因此需要打印第三步的条件是 startX < endX && startX < endY。第四步最复杂，首先 startX 要小于 endX，不然所有的数字都在一列，在第二步中就都打印完了。另外，这个圈中至少要有三行数字。如果只有一行数字，所有数字在第一步中打印完了；如果只有两行数字，所有数字在第一步和第三步也都打印完了。因此打印第四步需要的条件是 startY < endY – 1。

有了前面的分析，我们就能写出 PrintMatrixInCircle 的完整代码如下：

```
void PrintMatrixInCircle(int** numbers, int columns, int rows,
                         int startX, int startY)
{
    int endX = columns - 1 - startX;
    int endY = rows - 1 - startY;

    PrintARowIncreasingly(numbers, columns, rows, startY, startX, endX);

    if(startY < endY)
        PrintAColumnIncreasingly(numbers, columns, rows, endX, startY + 1, endY);

    if(startX < endX && startY < endY)
```

```
        PrintARowDecreasingly(numbers, columns, rows, endY, endX - 1, startX);

    if(startX < endX && startY < endY - 1)
        PrintAColumnDecreasingly(numbers, columns, rows, startX, endY - 1, startY + 1);
}
```

接下来我们考虑如何打印一行或者一列。这对我们来说不是一件很难的事情。以函数 PrintARowIncreasingly 为例，我们只需要一个循环，把行号为 startY，列号从 startX 到 endX 的所有数字依次从数组中取出来并逐个打印就行了，对应的代码是：

```
void PrintARowIncreasingly(int** numbers, int columns, int rows,
                           int y, int firstX, int lastX)
{
    for(int i = firstX; i <= lastX; ++i)
    {
        int number = *(*(numbers + y) + i);
    printf("%d\t", number);
    }
}
```

剩下的三个函数与此类似，代码依次如下：

```
void PrintAColumnIncreasingly(int** numbers, int columns, int rows,
                              int x, int firstY, int lastY)
{
    for(int i = firstY; i <= lastY; ++i)
    {
        int number = *(*(numbers + i) + x);
    printf("%d\t", number);
    }
}
void PrintARowDecreasingly(int** numbers, int columns, int rows,
                           int y, int firstX, int lastX)
{
    for(int i = firstX; i >= lastX; --i)
    {
        int number = *(*(numbers + y) + i);
    printf("%d\t", number);
    }
}
void PrintAColumnDecreasingly(int** numbers, int columns, int rows,
                              int x, int firstY, int lastY)
{
    for(int i = firstY; i >= lastY; --i)
    {
        int number = *(*(numbers + i) + x);
    printf("%d\t", number);
    }
```

```
}
```

(52)-C++面试题（1）

写在前面的话：由于与 C++语法相关的面试题，通常用很短的篇幅就能解释清楚，不适合写博客，因此本博客一直没有关注 C++的语法题。近期发现篇幅短的 C++题目刚好合适微博，于是开始在微博 http://t.sina.com.cn/zhedahht 和 http://t.163.com/zhedahht 上写 C++的系列面试题。感兴趣的读者可以关注我的微博，或者直接围观面试题每日一题系列。同时，我也将不定期整理一些经典的 C++面试题，发表到本博客上。

题目（一）：我们可以用 static 修饰一个类的成员函数，也可以用 const 修饰类的成员函数（写在函数的最后表示不能修改成员变量，不是指写在前面表示返回值为常量）。请问：能不能同时用 static 和 const 修饰类的成员函数？

分析：答案是不可以。C++编译器在实现 const 的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数 const this*。但当一个成员为 static 的时候，该函数是没有 this 指针的。也就是说此时 static 的用法和 static 是冲突的。

我们也可以这样理解：两者的语意是矛盾的。static 的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而 const 的作用是确保函数不能修改类的实例的状态，与类型的静态变量没有关系。因此不能同时用它们。

题目（二）：运行下面的代码，输出是什么？

```
class A
{
};

class B
{
public:
    B() {}
    ~B() {}
};

class C
{
public:
    C() {}
    virtual ~C() {}
};

int _tmain(int argc, _TCHAR* argv[])
{
    printf("%d, %d, %d\n", sizeof(A), sizeof(B), sizeof(C));
    return 0;
}
```

分析：答案是 1, 1, 4。class A 是一个空类型，它的实例不包含任何信息，本来求 sizeof 应该是 0。但当我们声明该类型的实例的时候，它必须在内存中占有一定的空间，否则无法使用这些实例。至于占用多少内存，由编译器决定。Visual Studio 2008 中每个空类型的实例占用一个 byte 的空间。

class B 在 class A 的基础上添加了构造函数和析构函数。由于构造函数和析构函数的调用与类型的实例无关（调用它们只需要知道函数地址即可），在它的实例中不需要增加任何信息。所以 sizeof(B)和 sizeof(A)一样，在 Visual Studio 2008 中都是 1。

class C 在 class B 的基础上把析构函数标注为虚拟函数。C++的编译器一旦发现一个类型中有虚拟函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向虚函数表的指针。在 32 位的机器上，一个指针占 4 个字节的空间，因此 sizeof(C)是 4。

题目（三）：运行下面中的代码，得到的结果是什么？

```
class A
{
private:
    int m_value;

public:
    A(int value)
    {
        m_value = value;
    }
    void Print1()
    {
        printf("hello world");
    }
    void Print2()
    {
        printf("%d", m_value);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A* pA = NULL;
    pA->Print1();
    pA->Print2();

    return 0;
}
```

分析：答案是 Print1 调用正常，打印出 hello world，但运行至 Print2 时，程序崩溃。调用 Print1 时，并不需要 pA 的地址，因为 Print1 的函数地址是固定的。编译器会给 Print1 传入一个 this 指针，该指针为 NULL，但在 Print1 中该 this 指针并没有用到。只要程序运行时没有访问不该访问的内存就不会出错，因此运行正常。在运行 print2 时，需要 this 指针才能得到 m_value 的值。由于此时 this 指针为 NULL，因此程序崩溃了。

题目（四）：运行下面中的代码，得到的结果是什么？

```
class A
{
private:
	int m_value;

public:
	A(int value)
	{
		m_value = value;
	}
	void Print1()
	{
		printf("hello world");
	}
	virtual void Print2()
	{
		printf("hello world");
	}
};

int _tmain(int argc, _TCHAR* argv[])
{
	A* pA = NULL;
	pA->Print1();
	pA->Print2();

	return 0;
}
```

分析：答案是 Print1 调用正常，打印出 hello world，但运行至 Print2 时，程序崩溃。Print1 的调用情况和上面的题目一样，不在赘述。由于 Print2 是虚函数。C++调用虚函数的时候，要根据实例（即 this 指针指向的实例）中虚函数表指针得到虚函数表，再从虚函数表中找到函数的地址。由于这一步需要访问实例的地址（即 this 指针），而此时 this 指针为空指针，因此导致内存访问出错。

题目（五）：静态成员函数能不能同时也是虚函数？

分析：答案是不能。调用静态成员函数不要实例。但调用虚函数需要从一个实例中指向虚函数表的指针以得到函数的地址，因此调用虚函数需要一个实例。两者相互矛盾。

(53)-C++/C#面试题（2）

写在前面的话：本次选用的 5 道题，是我微博（http://t.sina.com.cn/zhedahht 和

http://t.163.com/zhedahht）中#面试每日一题#系列的第 6 题到第 10 题。有合适的题目，我会继续收集 C/C++/C#的面试题，并不定期发表到博客和大家分享。

读者请在回复中告知在没有看答案之前能做对几题（得出正确的答案并能给出合理的解释），这样我就能知道博客中选用题目的难度是否适中。

题目（六）：运行下列 C++代码，输出什么？

```
struct Point3D
{
    int x;
    int y;
    int z;
};

int _tmain(int argc, _TCHAR* argv[])
{
    Point3D* pPoint = NULL;
    int offset = (int)(&(pPoint)->z);

    printf("%d", offset);
    return 0;
}
```

答案：输出 8。由于在 pPoint->z 的前面加上了取地址符号，运行到此时的时候，会在 pPoint 的指针地址上加 z 在类型 Point3D 中的偏移量 8。由于 pPoint 的地址是 0，因此最终 offset 的值是 8。

&(pPoint->z)的语意是求 pPoint 中变量 z 的地址（pPoint 的地址 0 加 z 的偏移量 8），并不需要访问 pPoint 指向的内存。只要不访问非法的内存，程序就不会出错。

题目（七）：运行下列 C++代码，输出什么？

```
class A
{
public:
    A()
    {
        Print();
    }
    virtual void Print()
    {
        printf("A is constructed.\n");
    }
};

class B: public A
{
public:
    B()
    {
```

```
        Print();
    }

    virtual void Print()
    {
        printf("B is constructed.\n");
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A* pA = new B();
    delete pA;

    return 0;
}
```

答案：先后打印出两行:A is constructed. B is constructed. 调用 B 的构造函数时，先会调用 B 的基类及 A 的构造函数。然后在 A 的构造函数里调用 Print。由于此时实例的类型 B 的部分还没有构造好，本质上它只是 A 的一个实例，他的虚函数表指针指向的是类型 A 的虚函数表。因此此时调用的 Print 是 A::Print，而不是 B::Print。接着调用类型 B 的构造函数，并调用 Print。此时已经开始构造 B，因此此时调用的 Print 是 B::Print。

同样是调用虚拟函数 Print，我们发现在类型 A 的构造函数中，调用的是 A::Print，在 B 的构造函数中，调用的是 B::Print。因此虚函数在构造函数中，已经失去了虚函数的动态绑定特性。

题目（八）：运行下列 C#代码，输出是什么？

```
namespace ChangesOnString
{
    class Program
    {
        static void Main(string[] args)
        {
            String str = "hello";
            str.ToUpper();
            str.Insert(0, " WORLD");

            Console.WriteLine(str);
        }
    }
}
```

答案：输出是 hello。由于在.NET 中，String 有一个非常特殊的性质：String 的实例的状态不能被改变。如果 String 的成员函数会修改实例的状态，将会返回一个新的 String 实例。改动只会出现在返回值中，而不会修改原来的实例。所以本题中输出仍然是原来的字符串值 hello。

如果试图改变 String 的内容，改变之后的值可以通过返回值拿到。用 StringBuilder 是更好的

选择，特别是要连续多次修改的时候。如果用 String 连续多次修改，每一次修改都会产生一个临时对象，开销太大。

题目（九）：在 C++和 C#中，struct 和 class 有什么不同？

答案：在 C++中，如果没有标明函数或者变量是的访问权限级别，在 struct 中，是 public 的；而在 class 中，是 private 的。

在 C#中，如果没有标明函数或者变量的访问权限级别，struct 和 class 中都是 private 的。struct 和 class 的区别是：struct 定义值类型，其实例在栈上分配内存；class 定义引用类型，其实例在堆上分配内存。

题目（十）：运行下图中的 C#代码，输出是什么？

```
namespace StaticConstructor
{
    class A
    {
        public A(string text)
        {
            Console.WriteLine(text);
        }
    }

    class B
    {
        static A a1 = new A("a1");
        A a2 = new A("a2");

        static B()
        {
            a1 = new A("a3");
        }

        public B()
        {
            a2 = new A("a4");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            B b = new B();
        }
    }
}
```

答案：打印出四行，分别是 a1、a3、a2、a4。

在调用类型 B 的代码之前先执行 B 的静态构造函数。静态函数先初始化类型的静态变量，再执行静态函数内的语句。因此先打印 a1 再打印 a3。接下来执行 B b = new B()，即调用 B 的普通构造函数。构造函数先初始化成员变量，在执行函数体内的语句，因此先后打印出 a2、a4。

(54)-C++/C#面试题（3）

写在前面的话：本次选用的 5 道题，是我微博（http://t.sina.com.cn/zhedahht 和 http://t.163.com/zhedahht）中#面试每日一题#系列的第 11 题到第 15 题。有合适的题目，我会继续收集 C/C++/C#的面试题，并不定期发表到博客和大家分享。

题目（11）：运行下图中的 C#代码，输出是什么？

```
namespace StringValueOrReference
{
    class Program
    {
        internal static 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.");
        }

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

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

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

            Console.WriteLine(text);
        }
    }
}
```

答案：输出两行。第一行是 The type String is reference type. 第二行是 hello。类型 String 的

定义是 public sealed class String {...}，既然是 class，那么 String 就是引用类型。
在方法 ModifyString 里，对 text 赋值一个新的字符串，此时改变的不是原来 text 的内容，而是把 text 指向一个新的字符串"world"。由于参数 text 没有加 ref 或者 out，出了方法之后，text 还是指向原来的字符串，因此输出仍然是"hello".
题目（12）：运行下图中的 C++代码，输出是什么？

```
#include <iostream>

class A
{
private:
    int n1;
    int n2;
public:
    A(): n2(0), n1(n2 + 2)
    {
    }

    void Print()
    {
        std::cout << "n1: " << n1 << ", n2: " << n2 << std::endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A a;
    a.Print();

    return 0;
}
```

答案：输出 n1 是一个随机的数字，n2 为 0。在 C++中，成员变量的初始化顺序与变量在类型中的申明顺序相同，而与它们在构造函数的初始化列表中的顺序无关。因此在这道题中，会首先初始化 n1，而初始 n1 的参数 n2 还没有初始化，是一个随机值，因此 n1 就是一个随机值。初始化 n2 时，根据参数 0 对其初始化，故 n2=0。
题目（13）：编译运行下图中的 C++代码，结果是什么？（A）编译错误；（B）编译成功，运行时程序崩溃；（C）编译运行正常，输出 10。请选择正确答案并分析原因。

```
#include <iostream>

class A
{
private:
    int value;

public:
```

```
    A(int n)
    {
        value = n;
    }

    A(A other)
    {
        value = other.value;
    }

    void Print()
    {
        std::cout << value << std::endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A a = 10;
    A b = a;
    b.Print();

    return 0;
}
```

答案：编译错误。在复制构造函数中传入的参数是 A 的一个实例。由于是传值，把形参拷贝到实参会调用复制构造函数。因此如果允许复制构造函数传值，那么会形成永无休止的递归并造成栈溢出。因此 C++的标准不允许复制构造函数传值参数，而必须是传引用或者常量引用。在 Visual Studio 和 GCC 中，都将编译出错。

题目（14）：运行下图中的 C++代码，输出是什么？

```
int SizeOf(char pString[])
{
    return sizeof(pString);
}

int _tmain(int argc, _TCHAR* argv[])
{
    char* pString1 = "google";
    int size1 = sizeof(pString1);
    int size2 = sizeof(*pString1);

    char pString2[100] = "google";
    int size3 = sizeof(pString2);
    int size4 = SizeOf(pString2);
```

```
    printf("%d, %d, %d, %d", size1, size2, size3, size4);

    return 0;
}
```

答案：4, 1, 100, 4。pString1 是一个指针。在 32 位机器上，任意指针都占 4 个字节的空间。*pString1 是字符串 pString1 的第一个字符。一个字符占一个字节。pString2 是一个数组，sizeof(pString2)是求数组的大小。这个数组包含 100 个字符，因此大小是 100 个字节。而在函数 SizeOf 中，虽然传入的参数是一个字符数组，当数组作为函数的参数进行传递时，数组就自动退化为同类型的指针。因此 size4 也是一个指针的大小，为 4.

题目（15）：运行下图中代码，输出的结果是什么？这段代码有什么问题？

```
#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A is created." << std::endl;
    }

    ~A()
    {
        std::cout << "A is deleted." << std::endl;
    }
};

class B : public A
{
public:
    B()
    {
        std::cout << "B is created." << std::endl;
    }

    ~B()
    {
        std::cout << "B is deleted." << std::endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A* pA = new B();
    delete pA;
```

```
    return 0;
}
```

答案：输出三行，分别是：A is created. B is created. A is deleted。用 new 创建 B 时，回调用 B 的构造函数。在调用 B 的构造函数的时候，会先调用 A 的构造函数。因此先输出 A is created. B is created.

接下来运行 delete 语句时，会调用析构函数。由于 pA 被声明成类型 A 的指针，同时基类 A 的析构函数没有标上 virtual，因此只有 A 的析构函数被调用到，而不会调用 B 的析构函数。由于 pA 实际上是指向一个 B 的实例的指针，但在析构的时候只调用了基类 A 的析构函数，却没有调用 B 的析构函数。这就是一个问题。如果在类型 B 中创建了一些资源，比如文件句柄、内存等，在这种情况下都得不到释放，从而导致资源泄漏。

(55)-不用＋、－、×、÷数字运算符做加法

题目：写一个函数，求两个整数的之和，要求在函数体内不得使用＋、－、×、÷。

分析：这又是一道考察发散思维的很有意思的题目。当我们习以为常的东西被限制使用的时候，如何突破常规去思考，就是解决这个问题的关键所在。

看到的这个题目，我的第一反应是傻眼了，四则运算都不能用，那还能用什么啊？可是问题总是要解决的，只能打开思路去思考各种可能性。首先我们可以分析人们是如何做十进制的加法的，比如是如何得出 5+17=22 这个结果的。实际上，我们可以分成三步的：第一步只做各位相加不进位，此时相加的结果是 12（个位数 5 和 7 相加不要进位是 2，十位数 0 和 1 相加结果是 1）；第二步做进位，5+7 中有进位，进位的值是 10；第三步把前面两个结果加起来，12+10 的结果是 22，刚好 5+17=22。

前面我们就在想，求两数之和四则运算都不能用，那还能用什么啊？对呀，还能用什么呢？对数字做运算，除了四则运算之外，也就只剩下位运算了。位运算是针对二进制的，我们也就以二进制再来分析一下前面的三步走策略对二进制是不是也管用。

5 的二进制是 101，17 的二进制 10001。还是试着把计算分成三步：第一步各位相加但不计进位，得到的结果是 10100（最后一位两个数都是 1，相加的结果是二进制的 10。这一步不计进位，因此结果仍然是 0）；第二步记下进位。在这个例子中只在最后一位相加时产生一个进位，结果是二进制的 10；第三步把前两步的结果相加，得到的结果是 10110，正好是 22。由此可见三步走的策略对二进制也是管用的。

接下来我们试着把二进制上的加法用位运算来替代。第一步不考虑进位，对每一位相加。0 加 0 与 1 加 1 的结果都 0，0 加 1 与 1 加 0 的结果都是 1。我们可以注意到，这和异或的结果是一样的。对异或而言，0 和 0、1 和 1 异或的结果是 0，而 0 和 1、1 和 0 的异或结果是 1。接着考虑第二步进位，对 0 加 0、0 加 1、1 加 0 而言，都不会产生进位，只有 1 加 1 时，会向前产生一个进位。此时我们可以想象成是两个数先做位与运算，然后再向左移动一位。只有两个数都是 1 的时候，位与得到的结果是 1，其余都是 0。第三步把前两个步骤的结果相加。如果我们定义一个函数 AddWithoutArithmetic，第三步就相当于输入前两步骤的结果来递归调用自己。

有了这些分析之后，就不难写出如下的代码了：

```
int AddWithoutArithmetic(int num1, int num2)
{
```

```
    if(num2 == 0)
        return num1;

    int sum = num1 ^ num2;
    int carry = (num1 & num2) << 1;

    return AddWithoutArithmetic(sum, carry);
}
```

之前我的系列博客中有这么一道题，求 1+2+…+n，要求不能使用乘除法、for、while、if、else、switch、case 等关键字以及条件判断语句（A?B:C）。刚兴趣的读者，可以到 http://zhedahht.blog.163.com/blog/static/254111742007291513142 2/看看。

(56)-C/C++/C#面试题（4）

问题（16）：运行如下的 C++代码，输出是什么？

```
class A
{
public:
    virtual void Fun(int number = 10)
    {
        std::cout << "A::Fun with number " << number;
    }
};

class B: public A
{
public:
    virtual void Fun(int number = 20)
    {
        std::cout << "B::Fun with number " << number;
    }
};

int main()
{
    B b;
    A &a = b;
    a.Fun();
}
```

答案：输出 B::Fun with number 10。由于 a 是一个指向 B 实例的引用，因此在运行的时候会调用 B::Fun。但缺省参数是在编译期决定的。在编译的时候，编译器只知道 a 是一个类型 a 的引用，具体指向什么类型在编译期是不能确定的，因此会按照 A::Fun 的声明把缺省参数

number 设为 10。

这一题的关键在于理解确定缺省参数的值是在编译的时候，但确定引用、指针的虚函数调用哪个类型的函数是在运行的时候。

问题（17）：运行如下的 C 代码，输出是什么？

```
char* GetString1()
{
    char p[] = "Hello World";
    return p;
}

char* GetString2()
{
    char *p = "Hello World";
    return p;
}


int _tmain(int argc, _TCHAR* argv[])
{
    printf("GetString1 returns: %s. \n", GetString1());
    printf("GetString2 returns: %s. \n", GetString2());

    return 0;
}
```

答案：输出两行，第一行 GetString1 returns: 后面跟的是一串随机的内容，而第二行 GetString2 returns: Hello World. 两个函数的区别在于 GetString1 中是一个数组，而 GetString2 中是一个指针。

当运行到 GetString1 时，p 是一个数组，会开辟一块内存，并拷贝"Hello World"初始化该数组。接着返回数组的首地址并退出该函数。由于 p 是 GetString1 内的一个局部变量，当运行到这个函数外面的时候，这个数组的内存会被释放掉。因此在_tmain 函数里再去访问这个数组的内容时，结果是随机的。

当运行到 GetString2 时，p 是一个指针，它指向的是字符串常量区的一个常量字符串。该常量字符串是一个全局的，并不会因为退出函数 GetString2 而被释放掉。因此在_tmain 中仍然根据 GetString2 返回的地址得到字符串"Hello World"。

问题（18）：运行下图中 C#代码，输出的结果是什么？

```
namespace StaticVariableInAppDomain
{
    [Serializable]
    internal class A : MarshalByRefObject
    {
        public static int Number;

        public void SetNumber(int value)
        {
```

```
            Number = value;
        }
    }

    [Serializable]
    internal class B
    {
        public static int Number;

        public void SetNumber(int value)
        {
            Number = value;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            String assamblyName = Assembly.GetEntryAssembly().FullName;
            AppDomain domain = AppDomain.CreateDomain("NewDomain");

            A.Number = 10;
            String nameOfA = typeof(A).FullName;
            A a = domain.CreateInstanceAndUnwrap(assamblyName, nameOfA) as A;
            a.SetNumber(20);
            Console.WriteLine("Number in class A is {0}", A.Number);

            B.Number = 10;
            String nameOfB = typeof(B).FullName;
            B b = domain.CreateInstanceAndUnwrap(assamblyName, nameOfB) as B;
            b.SetNumber(20);
            Console.WriteLine("Number in class B is {0}", B.Number);
        }
    }
}
```

答案：输出两行，第一行是 Number in class A is 10，而第二行是 Number in class B is 20。上述 C#代码先创建一个命名为 NewDomain 的应用程序域，并在该域中利用反射机制创建类型 A 的一个实例和类型 B 的一个实例。我们注意到类型 A 是继承自 MarshalByRefObject，而 B 不是。虽然这两个类型的结构一样，但由于基类不同而导致在跨越应用程序域的边界时表现出的行为将大不相同。

由于 A 继承 MarshalByRefObject，那么 a 实际上只是在缺省的域中的一个代理，它指向位于 NewDomain 域中的 A 的一个实例。当 a.SetNumber 时，是在 NewDomain 域中调用该方法，它将修改 NewDomain 域中静态变量 A.Number 的值并设为 20。由于静态变量在每个

应用程序域中都有一份独立的拷贝，修改 NewDomain 域中的静态变量 A.Number 对缺省域中的静态变量 A.NewDomain 没有任何影响。由于 Console.WriteLine 是在缺省的应用程序域中输出 A.Number，因此输出仍然是 10。

B 只从 Object 继承而来的类型，它的实例穿越应用程序域的边界时，将会完整地拷贝实例。在上述代码中，我们尽管试图在 NewDomani 域中生成 B 的实例，但会把实例 b 拷贝到缺省的域。此时，调用 b.SetNumber 也是在缺省的域上进行，它将修改缺省的域上的 A.Number 并设为 20。因此这一次输出的是 20。

问题（19）：运行下图中 C 代码，输出的结果是什么？

```
int _tmain(int argc, _TCHAR* argv[])
{
    char str1[] = "hello world";
    char str2[] = "hello world";

    char* str3 = "hello world";
    char* str4 = "hello world";

    if(str1 == str2)
        printf("str1 and str2 are same.\n");
    else
        printf("str1 and str2 are not same.\n");

    if(str3 == str4)
        printf("str3 and str4 are same.\n");
    else
        printf("str3 and str4 are not same.\n");

    return 0;
}
```

答案：输出两行。第一行是 str1 and str2 are not same，第二行是 str3 and str4 are same。

str1 和 str2 是两个字符串数组。我们会为它们分配两个长度为 12 个字节的空间，并把"hello world"的内容分别拷贝到数组中去。这是两个初始地址不同的数组，因此比较 str1 和 str2 的值，会不相同。str3 和 str4 是两个指针，我们无需为它们分配内存以存储字符串的内容，而只需要把它们指向"hello world“在内存中的地址就可以了。由于"hello world”是常量字符串，它在内存中只有一个拷贝，因此 str3 和 str4 指向的是同一个地址。因此比较 str3 和 str4 的值，会是相同的。

问题（20）：运行下图中 C#代码，输出的结果是什么？并请比较这两个类型各有什么特点，有哪些区别。

```
namespace Singleton
{
    public sealed class Singleton1
    {
        private Singleton1()
        {
            Console.WriteLine("Singleton1 constructed");
```

```
    }
    public static void Print()
    {
        Console.WriteLine("Singleton1 Print");
    }
    private static Singleton1 instance = new Singleton1();
    public static Singleton1 Instance
    {
        get
        {
            return instance;
        }
    }
}

public sealed class Singleton2
{
    Singleton2()
    {
        Console.WriteLine("Singleton2 constructed");
    }
    public static void Print()
    {
        Console.WriteLine("Singleton2 Print");
    }
    public static Singleton2 Instance
    {
        get
        {
            return Nested.instance;
        }
    }
    class Nested
    {
        static Nested() { }

        internal static readonly Singleton2 instance = new Singleton2();
    }
}

class Program
{
    static void Main(string[] args)
    {
```

```
            Singleton1.Print();
            Singleton2.Print();
        }
    }
}
```

答案：输出三行：第一行“Singleton1 constructed”，第二行“Singleton1 Print”，第三行“Singleton2 Print”。

当我们调用 Singleton1.Print 时，.NET 运行时会自动调用 Singleton1 的静态构造函数，并初始化它的静态变量。此时会创建一个 Singleton1 的实例，因此会调用它的构造函数。Singleton2 的实例是在 Nested 的静态构造函数里初始化的。只有当类型 Nested 被使用时，才回触发.NET 运行时调用它的静态构造函数。我们注意到我们只在 Singtон2.Instance 里面用到了 Nested。而在我们的代码中，只调用了 Singleton2.Print。因此不会创建 Singleton2 的实例，也不会调用它的构造函数。

这两个类型其实都是单例模式（Singleton）的实现。第二个实现 Singleton2 只在真的需要时，才会创建实例，而第一个实现 Singleton1 则不然。第二个实现在空间效率上更好。

(57)-O(n)时间的排序

题目：某公司有几万名员工，请完成一个时间复杂度为 O(n)的算法对该公司员工的年龄作排序，可使用 O(1)的辅助空间。

分析：排序是面试时经常被提及的一类题目，我们也熟悉其中很多种算法，诸如插入排序、归并排序、冒泡排序，快速排序等等。这些排序的算法，要么是 $O(n^2)$的，要么是 O(n*log*n)的。可是这道题竟然要求是 O(n)的，这里面到底有什么玄机呢？

题目特别强调是对一个公司的员工的年龄作排序。员工的数目虽然有几万人，但这几万员工的年龄却只有几十种可能。上班早的人一般也要等到将近二十岁才上班，一般人再晚到了六七十岁也不得不退休。

由于年龄总共只有几十种可能，我们可以很方便地统计出每一个年龄里有多少名员工。举个简单的例子，假设总共有 5 个员工，他们的年龄分别是 25、24、26、24、25。我们统计出他们的年龄，24 岁的有两个，25 岁的也有两个，26 岁的一个。那么我们根据年龄排序的结果就是：24、24、25、25、26，即在表示年龄的数组里写出两个 24、两个 25 和一个 26。

想明白了这种思路，我们就可以写出如下代码：

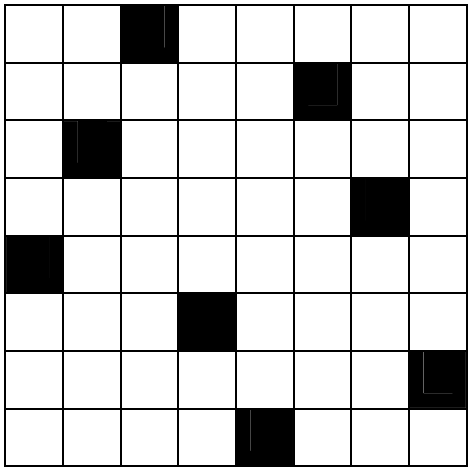
```
void SortAges(int ages[], int length)
{
    if(ages == NULL || length <= 0)
        return;

    const int oldestAge = 99;
    int timesOfAge[oldestAge + 1];

    for(int i = 0; i <= oldestAge; ++ i)
        timesOfAge[i] = 0;
```

```
    for(int i = 0; i < length; ++ i)
    {
        int age = ages[i];
        if(age < 0 || age > oldestAge)
            throw new std::exception("age out of range.");

        ++ timesOfAge[age];
    }

    int index = 0;
    for(int i = 0; i <= oldestAge; ++ i)
    {
        for(int j = 0; j < timesOfAge[i]; ++ j)
        {
            ages[index] = i;
            ++ index;
        }
    }
}
```

在上面的代码中，允许的范围是 0 到 99 岁。数组 timesOfAge 用来统计每个年龄出现的次数。某个年龄出现了多少次，就在数组 ages 里设置几次该年龄。这样就相当于给数组 ages 排序了。该方法用长度 100 的整数数组辅助空间换来了 O(n)的时间效率。由于不管对多少人的年龄作排序，辅助数组的长度是固定的 100 个整数，因此它的空间复杂度是个常数，即 O(1)。

(58)-八皇后问题

题目：在 8×8 的国际象棋上摆放八个皇后，使其不能相互攻击，即任意两个皇后不得处在同一行、同一列或者同一对角斜线上。下图中的每个黑色格子表示一个皇后，这就是一种符合条件的摆放方法。请求出总共有多少种摆法。



这就是有名的八皇后问题。解决这个问题通常需要用递归，而递归对编程能力的要求比较高。

因此有不少面试官青睐这个题目，用来考察应聘者的分析复杂问题的能力以及编程的能力。由于八个皇后的任意两个不能处在同一行，那么这肯定是每一个皇后占据一行。于是我们可以定义一个数组 ColumnIndex[8]，数组中第 i 个数字表示位于第 i 行的皇后的列号。先把 ColumnIndex 的八个数字分别用 0-7 初始化，接下来我们要做的事情就是对数组 ColumnIndex 做全排列。由于我们是用不同的数字初始化数组中的数字，因此任意两个皇后肯定不同列。我们只需要判断得到的每一个排列对应的八个皇后是不是在同一对角斜线上，也就是数组的两个下标 i 和 j，是不是 i-j==ColumnIndex[i]-Column[j] 或者 j-i==ColumnIndex[i]-ColumnIndex[j]。

关于排列的详细讨论，详见本系列博客的第 28 篇，《字符串的排列》，这里不再赘述。

接下来就是写代码了。思路想清楚之后，编码并不是很难的事情。下面是一段参考代码：

```
int g_number = 0;

void EightQueen()
{
    const int queens = 8;
    int ColumnIndex[queens];
    for(int i = 0; i < queens; ++ i)
        ColumnIndex[i] = i;

    Permutation(ColumnIndex, queens, 0);
}

void Permutation(int ColumnIndex[], int length, int index)
{
    if(index == length)
    {
        if(Check(ColumnIndex, length))
        {
            ++ g_number;
            PrintQueen(ColumnIndex, length);
        }
    }
    else
    {
        for(int i = index; i < length; ++ i)
        {
            int temp = ColumnIndex[i];
            ColumnIndex[i] = ColumnIndex[index];
            ColumnIndex[index] = temp;

            Permutation(ColumnIndex, length, index + 1);

            temp = ColumnIndex[index];
            ColumnIndex[index] = ColumnIndex[i];
```

```
                ColumnIndex[i] = temp;
            }
        }
}

bool Check(int ColumnIndex[], int length)
{
    for(int i = 0; i < length; ++ i)
    {
        for(int j = i + 1; j < length; ++ j)
        {
            if((i - j == ColumnIndex[i] - ColumnIndex[j])
                || (j - i == ColumnIndex[i] - ColumnIndex[j]))
            return false;
        }
    }

    return true;
}

void PrintQueen(int ColumnIndex[], int length)
{
    printf("Solution %d\n", g_number);

    for(int i = 0; i < length; ++i)
        printf("%d\t", ColumnIndex[i]);

    printf("\n");
}
```

(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》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)
```

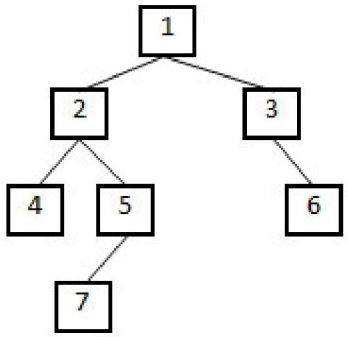
```
{
    if(string == NULL)
        return;

    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。

(60)-判断二叉树是不是平衡的

题目：**输入一棵二叉树的根结点，判断该树是不是平衡二叉树。如果某二叉树中任意结点的左右子树的深度相差不超过 1，那么它就是一棵平衡二叉树。**例如下图中的二叉树就是一

棵平衡二叉树：



在本系列博客的第 27 题，我们曾介绍过如何求二叉树的深度。有了求二叉树的深度的经验之后再解决这个问题，我们很容易就能想到一个思路：在遍历树的每个结点的时候，调用函数 TreeDepth 得到它的左右子树的深度。如果每个结点的左右子树的深度相差都不超过 1，按照定义它就是一棵平衡的二叉树。这种思路对应的代码如下：

```
bool IsBalanced(BinaryTreeNode* pRoot)
{
    if(pRoot == NULL)
        return true;

    int left = TreeDepth(pRoot->m_pLeft);
    int right = TreeDepth(pRoot->m_pRight);
    int diff = left - right;
    if(diff > 1 || diff < -1)
        return false;

    return IsBalanced(pRoot->m_pLeft) && IsBalanced(pRoot->m_pRight);
}
```

上面的代码固然简洁，但我们也要注意到由于一个节点会被重复遍历多次，这种思路的时间效率不高。例如在函数 IsBalance 中输入上图中的二叉树，首先判断根结点（值为 1 的结点）的左右子树是不是平衡结点。此时我们将往函数 TreeDepth 输入左子树根结点（值为 2 的结点），需要遍历结点 4、5、7。接下来判断以值为 2 的结点为根结点的子树是不是平衡树的时候，仍然会遍历结点 4、5、7。毫无疑问，重复遍历同一个结点会影响性能。接下来我们寻找不需要重复遍历的算法。

如果我们用后序遍历的方式遍历二叉树的每一个结点，在遍历到一个结点之前我们已经遍历了它的左右子树。只要在遍历每个结点的时候记录它的深度（某一结点的深度等于它到叶节点的路径的长度），我们就可以一边遍历一边判断每个结点是不是平衡的。下面是这种思路的参考代码：

```
bool IsBalanced(BinaryTreeNode* pRoot, int* pDepth)
{
    if(pRoot == NULL)
    {
        *pDepth = 0;
        return true;
    }
```

```
    int left, right;
    if(IsBalanced(pRoot->m_pLeft, &left)
        && IsBalanced(pRoot->m_pRight, &right))
    {
        int diff = left - right;
        if(diff <= 1 && diff >= -1)
        {
            *pDepth = 1 + (left > right ? left : right);
            return true;
        }
    }

    return false;
}
```

我们只需要给上面的函数传入二叉树的根结点以及一个表示结点深度的整形变量就可以了：

```
bool IsBalanced(BinaryTreeNode* pRoot)
{
    int depth = 0;
    return IsBalanced(pRoot, &depth);
}
```

在上面的代码中，我们用后序遍历的方式遍历整棵二叉树。在遍历某结点的左右子结点之后，我们可以根据它的左右子结点的深度判断它是不是平衡的，并得到当前结点的深度。当最后遍历到树的根结点的时候，也就判断了整棵二叉树是不是平衡二叉树了。