1.局部静态变量：函数中定义的静态变量，编译时分配内存，运行时第一次使用才进行初始化。C++0x标准实现了线程安全，而C++03标准没有

非局部静态变量：全局静态变量或类静态成员，编译时就分配内存并初始化，线程安全

2.合并数组（包括字符串）时，如果从前往后复制每个数字（或字符）需要重复移动多次，可以考虑从后往前复制，减少移动次数，从而提高效率。

3.递归的本质就是一个栈结构。如果一个问题能用栈结构更方便解决，那么也可以考虑用递归函数（实现更简单）。但要注意递归层次，防止过深而导致函数调用栈溢出。

基于递归实现的代码比基于循环的代码要简洁很多，但每次函数调用都需要在内存栈中分配空间以保存参数、返回地址及临时变量，压入和弹出数据都需要时间，因此递归实现的效率不如循环。递归方法通常都能通过循环并且存储中间变量的方法来实现。

4.不同排序算法适用场合不尽相同。快速排序虽然总体平均效率最好，但不是任何时候都是最优算法（比如数组本身已经排序，而每轮排序以最后一个数字作为比较的标准，此时效率只有O(n2)）。需要弄清楚排序应用的环境，有哪些约束条件！

5.位运算：把一个整数减去1之后，再和原来的整数做位与运算，得到的结果相当于把整数的二进制表示中最右边的一个1变成0.

位运算的效率比乘除法及求余运算的效率要高很多。右移运算表示除以2，左移表示乘以2，和0x1与运算判断奇数偶数。

6.斐波那契数列问题：递归方法的效率比循环的效率更低。

7.代码的完整性测试：功能测试、边界测试、负面测试。

8.由于计算机表示小数（包括float和double型小数）都有误差，不能直接用"=="判断两个小数是否相等，或者判断一个数是否为0,。如果两个小数的差的绝对值很小，如小于0.0000001，就可以认为它们相等。

9.关于n位的整数，并且没有限定n的取值范围，或者是输入任意大小的整数，那么就很有可能需要考虑大数问题。字符串是一个简单、有效的表示大数的方法。

10.解耦能够提高代码的重用性。使用函数指针可以提高算法扩展性。

11.提高代码的鲁棒性的有效途径就是进行防御性编程，即指预见在什么地方可能会出现问题，并为这些可能出现的问题制定处理方式。在看到问题的时候，要多问几个“如果不......那么......”这样的问题，例如：如果打开一个文件的时候发现文件不存在，那么该如何处理。在函数入口判断输入是否有效并对各种无效输入做好相应的处理。

（如果试图访问空指针指向的内存，程序会崩溃）

12.广度优先遍历一个有向图或者一棵树，都要用到队列。先把起始结点放入队列，每次从队列头部取出一个结点，遍历该结点后把它能到达的结点都依次放入队列。重复这个遍历过程，直到队列中的结点全部被遍历为止。

13.如果要判断多个字符是不是在某个字符串里出现过或者统计过多个字符在某个字符串中出现的次数，我们可以考虑基于数组创建一个简单的哈希表，这样可以用很小的空间消耗换来时间效率的提升。

14.查找算法通常用来在排序数组中查找一个数字，但通过修改可以查找重复数字的第一个和最后一个。

15.二叉树的遍历顺序影响了算法的效率。选择更合适的遍历顺序有时可以高效解决问题。

16.抽象建模：1）选择合理的数据结构来表述问题；2）分析模型中的内在规律，并用编程语言表述这种规律。

17.不使用新的变量，交换两个变量的值：

1)a = a + b; b = a - b; a = a - b; 2) a = a ^ b; b = a ^ b; a = a ^ b;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 单例模式（C++）\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Meyers Singleton

class Singleton

{

public:

static Singleton& Instance() //Instance()作为静态成员函数提供里全局访问点

{

static Singleton instance;

return instance;

}

private:

Singleton(); //这里将构造，析构，拷贝构造，赋值函数设为私有，杜绝了生成新例

~Singleton();

Singleton(const Singleton&);

Singleton& operator=(const Singleton&);

};

//双检测锁（线程安全）

class Singleton

{

public:

static Singleton& Instance() //Instance()作为静态成员函数提供里全局访问点

{

if(ps == NULL)

{

Lock(); //上锁

if(ps == NULL) //如果还未实例化，即可实例话，反之提供实例的引用

ps = new Singleton;

Unlock(); //解锁

}

return \*ps; //返回指针的话可能会误被 delete，返回引用安全一点

}

private:

Singleton(); //这里将构造，析构，拷贝构造，赋值函数设为私有，杜绝了生成新例

~Singleton();

Singleton(const Singleton&);

Singleton& operator=(const Singleton&);

static Singleton\* ps;

};

//源文件

Singleton\* Singleton::ps = NULL;