1. 解释= =和 Equals 的区别   
   　（1）= = 是一个运算符。   
   　（2）Equals则是string对象的方法，可以.（点）出来。   
   　（3）基本数据类型比较   
   　　 = =和Equals都比较两个值是否相等。相等为true 否则为false；   
   　（4）引用对象比较   
   　　 = =和Equals都是比较栈内存中的地址是否相等 。相等为true 否则为false；   
   　　   
   　　需注意几点：   
   　　a.string是一个特殊的引用类型。对于两个字符串的比较，不管是 = =和 Equals 这两者比较的都是字符串是否相同；   
   　　b.当你创建两个string对象时，内存中的地址是不相同的，你可以赋相同的值。   
   　　所以字符串的内容相同。引用地址不一定相同，（相同内容的对象地址不一定相同），但反过来却是肯定的；   
   　　c.基本数据类型比较(string 除外)= = 和 Equals 两者都是比较值；

2、比较分析三种拷贝

简单的来说就是，在有指针的情况下，浅拷贝只是增加了一个指针指向已经存在的内存，而深拷贝就是增加一个指针并且申请一个新的内存，使这个增加的指针指向这个新的内存，采用深拷贝的情况下，释放内存的时候就不会出现在浅拷贝时重复释放同一内存的错误！    
 编写C++程序中有时用到，操作符的重载。最能体现深层拷贝与浅层拷贝的，就是‘=’的重载。   
看下面一个简单的程序：   
  
class string   
{   
  
char \*m\_str;   
  
public:   
  
string(char \*s）   
  
{   
  
m\_str=s;   
  
}   
  
string()   
{};   
  
String & operator=(const string s)   
  
{   
  
m\_str=s.m\_str;   
  
return \*this   
}   
  
};   
  
int main()   
  
{   
  
string s1("abc"),s2;   
  
s2=s1;   
  
cout   
}   
  
 上面的 =重载其是就是实现了浅拷贝原因。是由于对象之中含有指针数据类型.s1,s2恰好指向同一各内存。所以是浅拷贝。而你如果修改一下原来的程序：   
string&operator=(const string&s)   
  
{   
  
if(strlen(m\_str)!=strlen(s.m\_str))   
  
m\_str=new char[strlen(s.m\_str)+1];   
  
if(\*this!=s)   
  
strcopy(m\_str,s.m\_str);   
  
return \*this;   
  
}   
 这样就实现了深拷贝，原因是你为被赋值对象申请了一个新的内存所以就是深拷贝。

1. 序列化和单例子模式进行对比结合

  为了使一个单例类变成可串行化的，仅仅在声明中添加 “implements Serializable”是不够的。因为一个串行化的对象在每次返串行化的时候，都会创建一个新的对象，而不仅仅是一个对原有对象的引用。为了防止这种情况，可以在单例类中加入readResolve 方法。  
  
  一般来说, 一个类实现了 Serializable接口, 我们就可以把它往内存地写再从内存里读出而"组装"成一个跟原来一模一样的对象. 不过当序列化遇到单例时,这里边就有了个问题: 从内存读出而组装的对象破坏了单例的规则. 单例是要求一个JVM中只有一个类对象的, 而现在通过反序列化,一个新的对象克隆了出来.  
  
  
如下例所示:    
  
  Java代码　　　    
     
public final class MySingleton implements Serializable {    
     private MySingleton() { }    
     private static final MySingleton INSTANCE = new MySingleton();    
     public static MySingleton getInstance() { return INSTANCE; }    
}    
    
 当把 MySingleton对象(通过getInstance方法获得的那个单例对象)序列化后再从内存中读出时, 就有一个全新但跟原来一样的MySingleton对象存在了. 那怎么来维护单例模式呢?这就要用到readResolve方法了.如下所示:   
Java代码　　    
  
    
public final class MySingleton implements Serializable{    
    private MySingleton() { }    
    private static final MySingleton INSTANCE = new MySingleton();    
    public static MySingleton getInstance() { return INSTANCE; }    
    private Object readResolve() throws ObjectStreamException {    
       // instead of the object we're on,    
       // return the class variable INSTANCE    
      return INSTANCE;    
   }    
}    
 这样当JVM从内存中反序列化地"组装"一个新对象时,就会自动调用这个 readResolve方法来返回我们指定好的对象了, 单例规则也就得到了保证 。  
  
 下面的这个例子 更能很好的说明readResove的必要性    
So far so good. Things get a little complicated when dealing with more than one instance however. To explain this, I'll show this using a type-safe enumeration. Keep in mind that Java 5's enum type automatically handles this readResolve case for you. Here is a nice little enumeration:     
    
    
public final class Sides {    
 private int value;    
 private Sides(int newVal) { value = newVal; }    
 private static final int LEFT\_VALUE = 1;    
 private static final int RIGHT\_VALUE = 2;    
 private static final int TOP\_VALUE = 3;    
 private static final int BOTTOM\_VALUE = 4;    
     
 public static final LEFT = new Sides(LEFT\_VALUE);    
 public static final RIGHT = new Sides(RIGHT\_VALUE);    
 public static final TOP = new Sides(TOP\_VALUE);    
 public static final BOTTOM = new Sides(BOTTOM\_VALUE);    
     
}    
    
    
Now, implementing serialization, the key to determining which instance to return is in inspecting what value is set on the object itself:     
    
    
public final class Sides implements Serializable {    
 private int value;    
 private Sides(int newVal) { value = newVal; }    
 private static final int LEFT\_VALUE = 1;    
 private static final int RIGHT\_VALUE = 2;    
 private static final int TOP\_VALUE = 3;    
 private static final int BOTTOM\_VALUE = 4;    
     
 public static final LEFT = new Sides(LEFT\_VALUE);    
 public static final RIGHT = new Sides(RIGHT\_VALUE);    
 public static final TOP = new Sides(TOP\_VALUE);    
 public static final BOTTOM = new Sides(BOTTOM\_VALUE);    
     
 private Object readResolve() throws ObjectStreamException {    
  // Switch on this instance's value to figure out which class variable    
  // this is meant to match    
  switch(value) {    
   case LEFT\_VALUE: return LEFT;    
   case RIGHT\_VALUE: return RIGHT;    
   case TOP\_VALUE: return TOP;    
   case BOTTOM\_VALUE: return BOTTOM;      
  }    
  return null;    
 }    
}  