拷贝构造函数是类中的一个特殊的成员函数，同时拷贝构造函数也是构造函数的一种，其函数名与类名相同，无返回类型，其作用也是为类的成员初始化以及为对象的构造分配存储空间，不同的是，拷贝构造函数的参数只有一个，且必须为该类对象的引用。

拷贝构造函数只有在使用一个对象初始化另外一个对象时才会使用，让我们从简单的说起：

int a = 10;

int b = a;

cout << a<< b << endl;

在该代码段中，使用整型变量a来初始化b，此时b的值即为a的值。同理：

[cpp] view plaincopy

```cpp
#include <iostream>
using namespace std;
class A{
public:
    A(int x)
    {
        a = x;
    }
    void show()
    {
        cout<<"a = "<<a<<endl;
    }

    ~A()
    {}
private:
    int a;
};

void main()
{
    int m = 10;
    A ob1(m);
    ob1.show();
    A ob2 = ob1;  //A ob2(ob1);
    ob2.show();
}
```

该代码段中，使用对象ob1来初始化对象ob2，此时ob2的成员a的值由ob1的成员a来初始化，即ob2.a = ob1.a;此过程是使用类的拷贝构造函数来完成的，但在上面的代码中,我们并没有看到拷贝构造函数，同样完成了复制工作，这又是为什么呢？因为当一个类没有自定义的拷贝构造函数的时候系统会自动提供一个默认的拷贝构造函数，来完成复制工作。

下面，我们为了说明情况，就普通情况而言(以上面的代码为例)，我们来自己定义一个与系统默认拷贝构造函数一样的拷贝构造函数，看看它的内部是如何工作的：

[cpp] view plaincopy

```cpp
#include <iostream>

using namespace std;

class A{
public:
    A(int x)  //构造函数
    {
        a = x;
    }
    void show()
    {
        cout<<"a = "<<a<<endl;
    }
    A(A &R_ob)//这里就是自定义的拷贝构造函数
    {
        cout << "使用拷贝构造函数" << endl;
        a = R_ob.a;//这句如果去掉就不能完成复制工作
    }

    ~A()
    {}
private:
    int a;
};

void main()
{
    int m = 10;
    A ob1(m);
    ob1.show();
    A ob2 = ob1;  //A ob2(ob1);
```

```
33.     ob2.show();
34. }
```

上面代码中的A(A &R_ob)就是我们自定义的拷贝构造函数，拷贝构造函数的名称必须与类名称一致，函数的形式参数是本类型的一个引用，且必须是引用。

当用一个已经初始化过了的自定义类类型对象去初始化另一个新构造的对象的时候，拷贝构造函数就会被自动调用，如果你没有自定义拷贝构造函数的时候系统将会提供给一个默认的拷贝构造函数来完成这个过程，上面代码的复制核心语句就是通过A(A &R_ob)拷贝构造函数内的a = R_ob.a;语句完成的。如果去掉这句代码，那么b对象的a属性将得到一个未知的随机值。

下面我们来讨论一下关于浅拷贝和深拷贝的问题。就上面的代码情况而言，很多人会问到，既然系统会自动提供一个默认的拷贝构造函数来处理复制，那么我们没有必要去自定义拷贝构造函数啊，没错，就普通情况而言这的确是没有必要的，但在某些状况下，类体内的成员是需要动态开辟堆内存的，如果我们不自定义拷贝构造函数而让系统自己处理，那么就会导致堆内存的所属权产生混乱。试想一下，已经开辟的一段堆地址原来是属于对象a的，由于复制过程发生，b对象取得是a已经开辟的堆地址，一旦程序产生析构，释放堆的时候，计算机是不可能清楚这段地址是真正属于谁的，当连续发生两次析构的时候就出现了运行错误。

[cpp] view plaincopy

```cpp
#include <iostream>
#include <string.h>

using namespace std;

class A{
public:
    A(char *n,char *ptr)
    {
        strcpy(name,n);  //静态空间的赋值
        address = new char[strlen(ptr) + 1];  //动态空间申请之后的赋值
        if(address != NULL)
            strcpy(address,ptr);
    }
    void show()
    {
        cout<<"Name is:"<<name<<endl;
        cout<<"Address is:"<<address<<endl;
    }
```

```
20.     A(A &R_ob)//这里就是自定义的拷贝构造函数
21.     {
22.         cout<<"使用拷贝构造函数"<<endl;
23.         strcpy(name,R_ob.name);
24.         address = new char[strlen(R_ob.address) + 1];  //动态空间申请之后
的赋值
25.         if(address != NULL)
26.             strcpy(address,R_ob.address);
27.     }
28.     ~A()
29.     {
30.         cout<<"调用析构函数"<<endl;
31.         delete []address;
32.     }
33. private:
34.     char name[20];
35.     char *address;
36. };
37.
38. void main()
39. {
40.     A ob1("张三","南昌");
41.     ob1.show();
42.     A ob2 = ob1;  //A ob2(ob1);
43.     ob2.show();
44. }
```

上面代码就演示了深拷贝的问题,对对象ob2的address属性采取了新开辟内存的方式避免了内存归属不清所导致析构释放空间时候的错误，最后必须提一下，对于上面的程序解释并不多，就是希望读者本身运行程序观察变化，进而深刻理解。（读者可以试着将拷贝构造函数注释掉运行程序看结果，也可采用单步调试的方式观察程序的执行过程）

深拷贝和浅拷贝的定义可以简单理解成：如果一个类拥有资源（堆，或者是其它系统资源），当这个类的对象发生复制过程的时候，这个过程就可以叫做深拷贝，反之对象存在资源但复制过程并未复制资源的情况视为浅拷贝。

浅拷贝资源后在释放资源的时候会产生资源归属不清的情况导致程序运行出错，这点尤其需要注意。