拷贝构造函数

C++中对象的复制就如同“克隆”，用一个已有的对象快速地复制出多个完全相同的对象。一般而言，以下三种情况都会使用到对象的复制：

（1）建立一个新对象，并用另一个同类的已有对象对新对象进行初始化，例如：

```
  class Rect
{
private:
       int width;
       int height;
};
Rect rect1;
Rect rect2(rect1);   // 使用 rect1 初始化 rect2，此时会进行对象的复制
```

（2）当函数的参数为类的对象时，这时调用此函数时使用的是值传递，也会产生对象的复制，例如：

```
  void fun1(Rect rect)
{
       ...
}
int main()
{
       Rect rect1;
       fun1(rect1);        // 此时会进行对象的复制
       return 0;
}
```

（3）函数的返回值是类的对象时，在函数调用结束时，需要将函数中的对象复制一个临时对象并传给改函数的调用处，例如：

```
  Rect fun2()
{
       Rect rect;
       return rect;
}
int main()
{
       Rect rect1;
       rect1=fun2();
       // 在 fun2 返回对象时，会执行对象复制，复制出一临时对象，
       // 然后将此临时对象“赋值”给 rect1
       return 0;
}
```

对象的复制都是通过一种特殊的构造函数来完成的，这种特殊的构造函数就是拷贝构造函数（copy constructor，也叫复制构造函数）。拷贝构造函数在大多数情况下都很简单，甚至在我们都不知道它存在的情况下也能很好发挥作用，但是在一些特殊情况下，特别

是在对象里有动态成员的时候，就需要我们特别小心地处理拷贝构造函数了。下面我们就来看看拷贝构造函数的使用。

一、默认拷贝构造函数

很多时候在我们都不知道拷贝构造函数的情况下，传递对象给函数参数或者函数返回对象都能很好的进行，这是因为编译器会给我们自动产生一个拷贝构造函数，这就是“默认拷贝构造函数”，这个构造函数很简单，仅仅使用“老对象”的数据成员的值对“新对象”的数据成员一一进行赋值，它一般具有以下形式：

```
  Rect::Rect(const Rect& r)
{
       width = r.width;
       height = r.height;
}
```

当然，以上代码不用我们编写，编译器会为我们自动生成。但是如果认为这样就可以解决对象的复制问题，那就错了，让我们来考虑以下一段代码：

```
  class Rect
{
public:
       Rect()            // 构造函数，计数器加 1
       {
              count++;
       }
       ~Rect()          // 析构函数，计数器减 1
       {
              count--;
       }
       static int getCount()              // 返回计数器的值
       {
              return count;
       }
private:
       int width;
       int height;
       static int count;              // 一静态成员做为计数器
};
int Rect::count = 0;               // 初始化计数器
int main()
{
       Rect rect1;
       cout<<"The count of Rect: "<<Rect::getCount()<<endl;
       Rect rect2(rect1);     // 使用 rect1 复制 rect2，此时应该有两个对象
       cout<<"The count of Rect: "<<Rect::getCount()<<endl;
       return 0;
}
```

这段代码对前面的类进行了一下小小的修改，加入了一个静态成员，目的是进行计数，统计创建的对象的个数，在每个对象创建时，通过构造函数进行递增，在销毁对象时，通过析构函数进行递减。在主函数中，首先创建对象 rect1，输出此时的对象个数，然后使用 rect1 复制出对象 rect2，再输出此时的对象个数，按照理解，此时应该有两个对象存在，但实际程序运行时，输出的都是 1，反应出只有 1 个对象。此外，在销毁对象时，由于会调用销毁两个对象，类的析构函数会调用两次，此时的计数器将变为负数。出现这些问题最根本就在于在复制对象时，计数器没有递增，解决的办法就是重新编写拷贝构造函数，在拷贝构造函数中加入对计数器的处理，形成的拷贝构造函数如下：

```
  class Rect
{
public:
	Rect()			// 构造函数，计数器加 1
	{
		count++;
	}
	Rect(const Rect& r)	// 拷贝构造函数
	{
		width = r.width;
		height = r.height;
		count++;			// 计数器加 1
	}
	~Rect()			// 析构函数，计数器减 1
	{
		count--;
	}
	static int getCount()	// 返回计数器的值
	{
		return count;
	}
private:
	int width;
	int height;
	static int count;		// 一静态成员做为计数器
};
```

自己编写拷贝构造函数又可以分为两种情况——浅拷贝与深拷贝。

二、浅拷贝

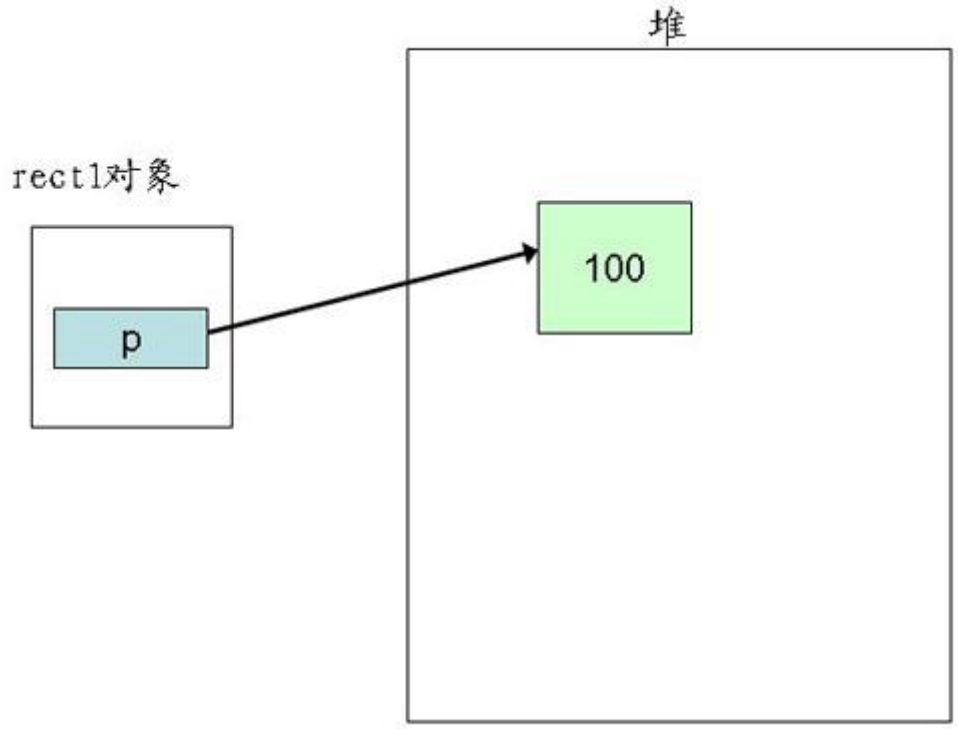
所谓浅拷贝，指的是在对象复制时，只是对对象中的数据成员进行简单的赋值，上面的例子都是属于浅拷贝的情况，默认拷贝构造函数执行的也是浅拷贝。大多情况下“浅拷贝”已经能很好地工作了，但是一旦对象存在了动态成员，那么浅拷贝就会出问题了，让我们考虑如下一段代码：

```
  class Rect
{
public:
```

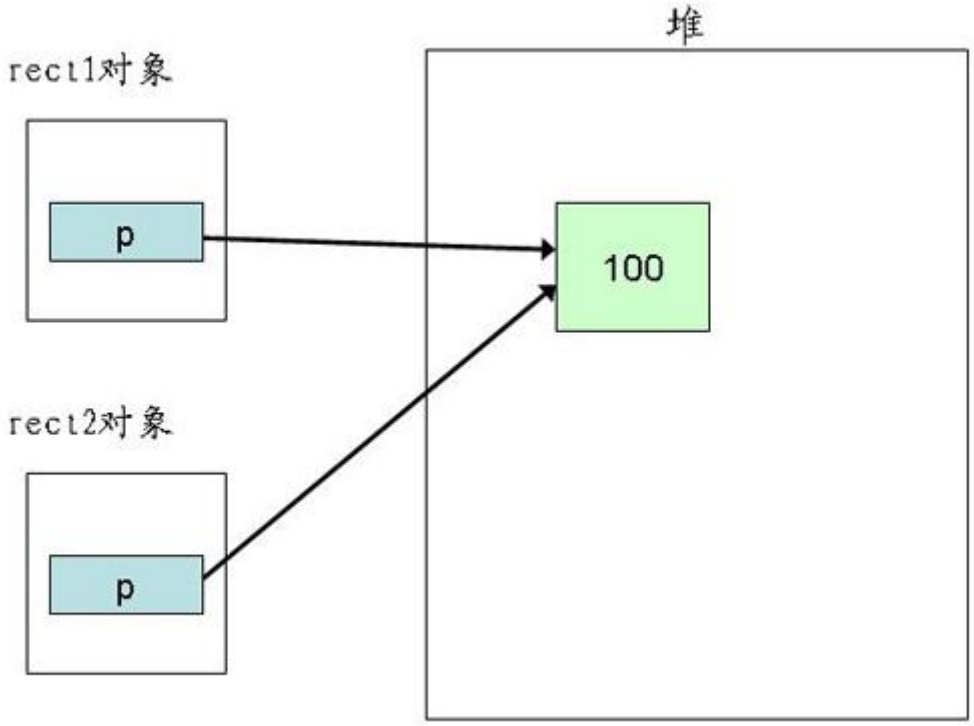
```
        Rect()              // 构造函数，p 指向堆中分配的一空间
        {
                p = new int(100);
        }
        ~Rect()             // 析构函数，释放动态分配的空间
        {
                if(p != NULL)
                {
                        delete p;
                }
        }
private:
        int width;
        int height;
        int *p;             // 一指针成员
};
int main()
{
        Rect rect1;
        Rect rect2(rect1);      // 复制对象
        return 0;
}
```

在这段代码运行结束之前，会出现一个运行错误。原因就在于在进行对象复制时，对于动态分配的内容没有进行正确的操作。我们来分析一下：

在运行定义 rect1 对象后，由于在构造函数中有一个动态分配的语句，因此执行后的内存情况大致如下：



在使用 rect1 复制 rect2 时，由于执行的是浅拷贝，只是将成员的值进行赋值，所以此时 rect1.p 和 rect2.p 具有相同的值，也即这两个指针指向了堆里的同一个空间，如下图所示：



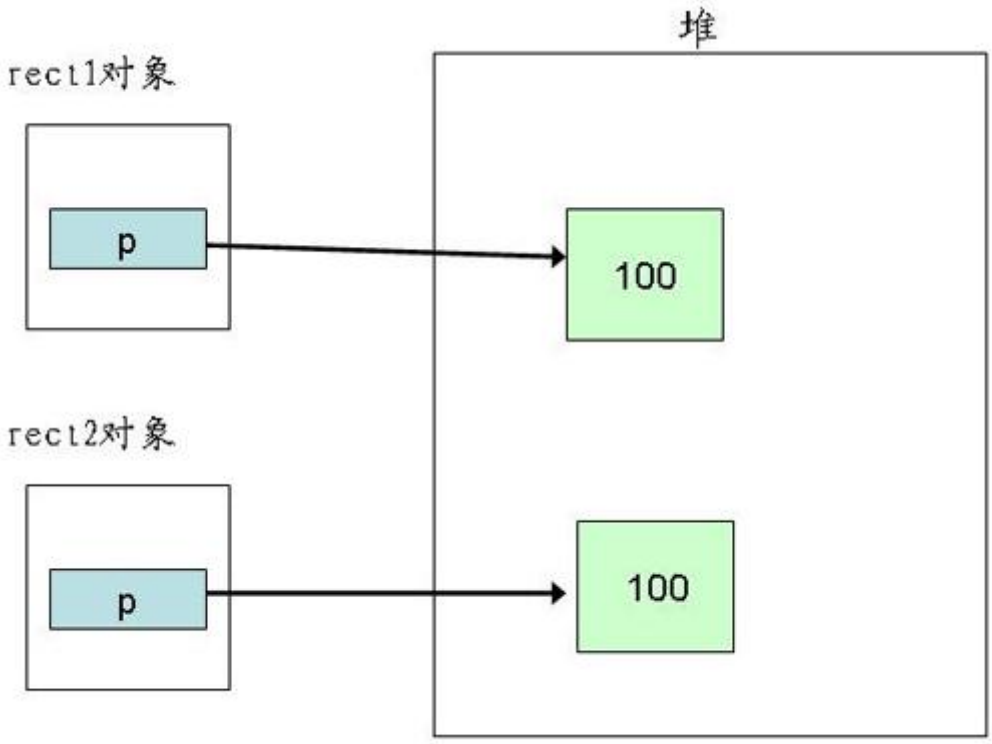
当然，这不是我们所期望的结果，在销毁对象时，两个对象的析构函数将对同一个内存空间释放两次，这就是错误出现的原因。我们需要的不是两个 p 有相同的值，而是两个 p 指向的空间有相同的值，解决办法就是使用“深拷贝”。

三、深拷贝

在“深拷贝”的情况下，对于对象中动态成员，就不能仅仅简单地赋值了，而应该重新动态分配空间，如上面的例子就应该按照如下的方式进行处理：

```
  class Rect
{
public:
	Rect()			// 构造函数，p 指向堆中分配的一空间
	{
		p = new int(100);
	}
	Rect(const Rect& r)
	{
		width = r.width;
		height = r.height;
		p = new int;		// 为新对象重新动态分配空间
		*p = *(r.p);
	}
	~Rect()			// 析构函数，释放动态分配的空间
	{
		if(p != NULL)
		{
			delete p;
		}
	}
private:
	int width;
	int height;
	int *p;			// 一指针成员
};
```

此时，在完成对象的复制后，内存的一个大致情况如下：



此时 rect1 的 p 和 rect2 的 p 各自指向一段内存空间，但它们指向的空间具有相同的内容，这就是所谓的“深拷贝”。

此外，在与“对象的复制”很类似的“对象的赋值”的情况下，也会出现同样的问题。在“对象的赋值”一文中再来讨论此问题。

通过对对象复制的分析，我们发现对象的复制大多在进行“值传递”时发生，这里有一个小技巧可以防止按值传递——声明一个私有拷贝构造函数。甚至不必去定义这个拷贝构造函数，这样因为拷贝构造函数是私有的，如果用户试图按值传递或函数返回该类对象，将得到一个编译错误，从而可以避免按值传递或返回对象。