**一. 什么是拷贝构造函数**

首先对于普通类型的对象来说，它们之间的复制是很简单的，例如：

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

int a = 100;

int b = a;

  
而类对象与普通对象不同，类对象内部结构一般较为复杂，存在各种成员变量。  
下面看一个类对象拷贝的简单例子。

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

#include <iostream>

using namespace std;

class CExample {

private:

　int a;

public:

//构造函数

　CExample(int b)

　{ a = b;}

//一般函数

　void Show ()

　{

cout<<a<<endl;

}

};

int main()

{

　CExample A(100);

　CExample B = A; //注意这里的对象初始化要调用拷贝构造函数，而非赋值

　 B.Show ();

　return 0;

}



运行程序，屏幕输出100。从以上代码的运行结果可以看出，系统为对象 B 分配了内存并完成了与对象 A 的复制过程。就类对象而言，相同类型的类对象是通过拷贝构造函数来完成整个复制过程的。

下面举例说明拷贝构造函数的工作过程。

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

#include <iostream>

using namespace std;

class CExample {

private:

int a;

public:

//构造函数

CExample(int b)

{ a = b;}

//拷贝构造函数

CExample(const CExample& C)

{

a = C.a;

}

//一般函数

void Show ()

{

cout<<a<<endl;

}

};

int main()

{

CExample A(100);

CExample B = A; // CExample B(A); 也是一样的

B.Show ();

return 0;

}

  
CExample(const CExample& C)　就是我们自定义的拷贝构造函数。可见，拷贝构造函数是一种**特殊的构造函数**，函数的名称必须和类名称一致，它必须的一个参数是本类型的一个**引用变量**。

**二. 拷贝构造函数的调用时机**

在C++中，下面三种对象需要调用拷贝构造函数！  
**1. 对象以值传递的方式传入函数参数**

**[c-sharp]** [**view plaincopyprint?**](http://blog.csdn.net/lwbeyond/article/details/6202256)

**class CExample**

**{**

**private:**

**int a;**

**public:**

**//构造函数**

**CExample(int b)**

**{**

**a = b;**

**cout<<"creat: "<<a<<endl;**

**}**

**//拷贝构造**

**CExample(const CExample& C)**

**{**

**a = C.a;**

**cout<<"copy"<<endl;**

**}**

**//析构函数**

**~CExample()**

**{**

**cout<< "delete: "<<a<<endl;**

**}**

**void Show ()**

**{**

**cout<<a<<endl;**

**}**

**};**

**//全局函数，传入的是对象**

**void g\_Fun(CExample C)**

**{**

**cout<<"test"<<endl;**

**}**

**int main()**

**{**

**CExample test(1);**

**//传入对象**

**g\_Fun(test);**

**return 0;**

**}**

****调用g\_Fun()时，会产生以下几个重要步骤：  
(1).test对象传入形参时，会先会产生一个临时变量，就叫 C 吧。  
(2).然后调用拷贝构造函数把test的值给C。 整个这两个步骤有点像：CExample C(test);  
(3).等g\_Fun()执行完后, 析构掉 C 对象。

**2. 对象以值传递的方式从函数返回**

**[c-sharp]** [**view plaincopyprint?**](http://blog.csdn.net/lwbeyond/article/details/6202256)

**class CExample**

**{**

**private:**

**int a;**

**public:**

**//构造函数**

**CExample(int b)**

**{**

**a = b;**

**}**

**//拷贝构造**

**CExample(const CExample& C)**

**{**

**a = C.a;**

**cout<<"copy"<<endl;**

**}**

**void Show ()**

**{**

**cout<<a<<endl;**

**}**

**};**

**//全局函数**

**CExample g\_Fun()**

**{**

**CExample temp(0);**

**return temp;**

**}**

**int main()**

**{**

**g\_Fun();**

**return 0;**

**}**

****当g\_Fun()函数执行到return时，会产生以下几个重要步骤：  
(1). 先会产生一个临时变量，就叫XXXX吧。  
(2). 然后调用拷贝构造函数把temp的值给XXXX。整个这两个步骤有点像：CExample XXXX(temp);  
(3). 在函数执行到最后先析构temp局部变量。  
(4). 等g\_Fun()执行完后再析构掉XXXX对象。

**3. 对象需要通过另外一个对象进行初始化；**

**[c-sharp]** [**view plaincopyprint?**](http://blog.csdn.net/lwbeyond/article/details/6202256)

**CExample A(100);**

**CExample B = A;**

**// CExample B(A);**

****

后两句都会调用拷贝构造函数。

**三. 浅拷贝和深拷贝**

**1. 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：

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

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：

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

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，加入了一个静态成员，目的是进行计数。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。

说白了，就是拷贝构造函数没有处理静态数据成员。

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，我们重新编写拷贝构造函数，如下：

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

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

};



**2. 浅拷贝**

所谓浅拷贝，指的是在对象复制时，只对对象中的数据成员进行简单的赋值，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：

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

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

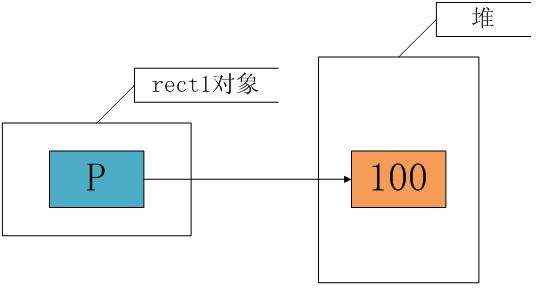
return 0;

}

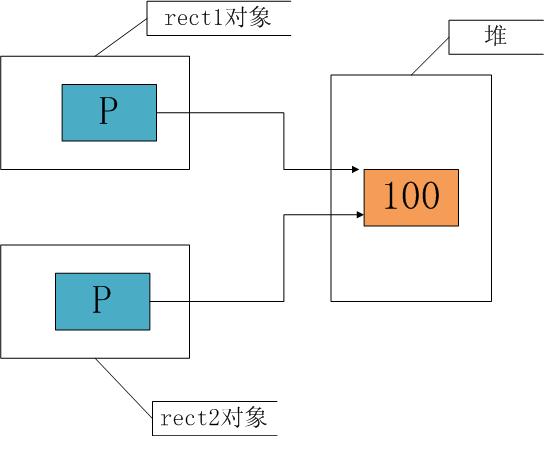


在这段代码运行结束之前，会出现一个运行错误。原因就在于在进行对象复制时，对于动态分配的内容没有进行正确的操作。我们来分析一下：

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，这时rect1.p = rect2.p，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”。

**3. 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：

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

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;

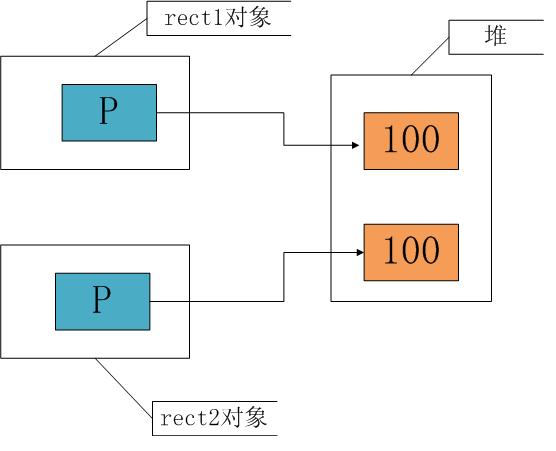
int height;

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”。

**3. 防止默认拷贝发生**

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。

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

// 防止按值传递

class CExample

{

private:

int a;

public:

//构造函数

CExample(int b)

{

a = b;

cout<<"creat: "<<a<<endl;

}

private:

//拷贝构造，只是声明

CExample(const CExample& C);

public:

~CExample()

{

cout<< "delete: "<<a<<endl;

}

void Show ()

{

cout<<a<<endl;

}

};

//全局函数

void g\_Fun(CExample C)

{

cout<<"test"<<endl;

}

int main()

{

CExample test(1);

//g\_Fun(test); 按值传递将出错

return 0;

}



**四. 拷贝构造函数的几个细节**

**1. 拷贝构造函数里能调用private成员变量吗?  
解答：**这个问题是在网上见的，当时一下子有点晕。其时从名子我们就知道拷贝构造函数其时就是一个特殊的**构造函数**，操作的还是自己类的成员变量，所以不受private的限制。

**2. 以下函数哪个是拷贝构造函数,为什么?**

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

X::X(const X&);

X::X(X);

X::X(X&, int a=1);

X::X(X&, int a=1, int b=2);

 **解答：**对于一个类X, 如果一个构造函数的第一个参数是下列之一:  
a) X&  
b) const X&  
c) volatile X&  
d) const volatile X&  
且没有其他参数或其他参数都有默认值,那么这个函数是拷贝构造函数.

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

X::X(const X&); //是拷贝构造函数

X::X(X&, int=1); //是拷贝构造函数

X::X(X&, int a=1, int b=2); //当然也是拷贝构造函数



**3. 一个类中可以存在多于一个的拷贝构造函数吗?  
解答：**类中可以存在超过一个拷贝构造函数。

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

class X {

public:

X(const X&); // const 的拷贝构造

X(X&); // 非const的拷贝构造

};

  
注意,如果一个类中只存在一个参数为 X& 的拷贝构造函数,那么就不能使用const X或volatile X的对象实行拷贝初始化.

**[c-sharp]** [view plaincopyprint?](http://blog.csdn.net/lwbeyond/article/details/6202256)

class X {

public:

X();

X(X&);

};

const X cx;

X x = cx; // error

  
如果一个类中没有定义拷贝构造函数,那么编译器会自动产生一个默认的拷贝构造函数。  
这个默认的参数可能为 X::X(const X&)或X::X(X&),由编译器根据上下文决定选择哪一个。