C++拷贝构造函数（复制构造函数）详解

复制构造函数是构造函数的一种，也称拷贝构造函数，它只有一个参数，参数类型是本类的引用。

复制构造函数的参数可以是 const 引用，也可以是非 const 引用。 一般使用前者，这样既能以常量对象（初始化后值不能改变的对象）作为参数，也能以非常量对象作为参数去初始化其他对象。一个类中写两个复制构造函数，一个的参数是 const 引用，另一个的参数是非 const 引用，也是可以的。

如果类的设计者不写复制构造函数，编译器就会自动生成复制构造函数。大多数情况下，其作用是实现从源对象到目标对象逐个字节的复制，即使得目标对象的每个成员变量都变得和源对象相等。编译器自动生成的复制构造函数称为“默认复制构造函数”。

注意，默认构造函数（即无参构造函数）不一定存在，但是复制构造函数总是会存在。

下面是一个复制构造函数的例子。

```
01.  #include<iostream >
02.  using namespace std;
03.  class Complex
04.  {
05.  public:
06.      double real, imag;
07.      Complex(double r, double i) {
08.          real= r; imag = i;
09.      }
10.  };
11.  int main(){
12.      Complex c1(1, 2);
13.      Complex c2 (c1);  //用复制构造函数初始化c2
14.      cout<<c2.real<<","<<c2.imag;  //输出 1,2
15.      return 0;
16.  }
```

第 13 行给出了初始化 c2 的参数，即 c1。只有编译器自动生成的那个默认复制构造函数的参数才能和 c1 匹配，因此，c2 就是以 c1 为参数，调用默认复制构造函数进行初始化的。初始化的结果是 c2 成为 c1 的复制品，即 c2 和 c1 每个成员变量的值都相等。

如果编写了复制构造函数，则默认复制构造函数就不存在了。下面是一个非默认复制构造函数的例子。

```
01.  #include<iostream>
02.  using namespace std;
03.  class Complex{
```

```
04.  public:
05.      double real, imag;
06.      Complex(double r,double i){
07.          real = r; imag = i;
08.      }
09.      Complex(const Complex & c){
10.          real = c.real; imag = c.imag;
11.          cout<<"Copy Constructor called"<<endl ;
12.      }
13.  };
14.
15.  int main(){
16.      Complex c1(1, 2);
17.      Complex c2 (c1);  //调用复制构造函数
18.      cout<<c2.real<<","<<c2.imag;
19.      return 0;
20.  }
```

程序的输出结果是：
Copy Constructor called
1,2

第 9 行，复制构造函数的参数加不加 const 对本程序來说都一样。但加上 const 是更好的做法，这样复制构造函数才能接受常量对象作为参数，即才能以常量对象作为参数去初始化别的对象。

第 17 行，就是以 c1 为参数调用第 9 行的那个复制构造函数初始化的。该复制构造函数执行的结果是使 c2 和 c1 相等，此外还输出 `Copy Constructor called` 。

可以想象，如果将第 10 行删去或改成 `real = 2*c.real; imag = imag + 1;` ，那么 c2 的值就不会等于 c1 了。也就是说，自己编写的复制构造函数并不一定要做复制的工作（如果只做复制工作，那么使用编译器自动生成的默认复制构造函数就行了）。但从习惯上来讲，复制构造函数还是应该完成类似于复制的工作为好，在此基础上还可以根据需要做些别的操作。

构造函数不能以本类的对象作为唯一参数，以免和复制构造函数相混淆。例如，不能写如下构造函数：

```
Complex (Complex c) {...}
```

复制构造函数被调用的三种情况

复制构造函数在以下三种情况下会被调用。

1) 当用一个对象去初始化同类的另一个对象时，会引发复制构造函数被调用。例如，下面的两条语句都会引发复制构造函数的调用，用以初始化 c2。

```
Complex c2(c1);
Complex c2 = c1;
```

这两条语句是等价的。

注意，第二条语句是初始化语句，不是赋值语句。赋值语句的等号左边是一个早已有定义的变量，赋值语句不会引发复制构造函数的调用。例如：

```
Complex c1, c2; c1 = c2 ;
c1=c2;
```

这条语句不会引发复制构造函数的调用，因为 c1 早已生成，已经初始化过了。

2) 如果函数 F 的参数是类 A 的对象，那么当 F 被调用时，类 A 的复制构造函数将被调用。换句话说，作为形参的对象，是用复制构造函数初始化的，而且调用复制构造函数时的参数，就是调用函数时所给的实参。

```
#include<iostream>
using namespace std;
class A{
public:
    A(){};
    A(A & a){
        cout<<"Copy constructor called"<<endl;
    }
};

void Func(A a){ }

int main(){
    A a;
    Func(a);
    return 0;
}
```

程序的输出结果为：
Copy constructor called

这是因为 Func 函数的形参 a 在初始化时调用了复制构造函数。

前面说过，函数的形参的值等于函数调用时对应的实参，现在可以知道这不一定是正确的。如果形参是一个对象，那么形参的值是否等于实参，取决于该对象所属的类的复制构造函数是如何实现的。例如上面的例子，Func 函数的形参 a 的值在进入函数时是随机的，未必等于实参，因为复制构造函数没有做复制的工作。

以对象作为函数的形参，在函数被调用时，生成的形参要用复制构造函数初始化，这会带来时间上的开销。如果用对象的引用而不是对象作为形参，就没有这个问题了。但是以引用作为形参有一定的风险，因为这种情况下如果形参的值发生改变，实参的值也会跟着改变。

如果要确保实参的值不会改变，又希望避免复制构造函数带来的开销，解决办法就是将形参声明为对象的 const 引用。例如：

```
01.  void Function(const Complex & c)
02.  {
03.      ...
04.  }
```

这样，Function 函数中出现任何有可能导致 c 的值被修改的语句，都会引发编译错误。

思考题：在上面的 Function 函数中，除了赋值语句，还有什么语句有可能改变 c 的值？例如，是否允许通过 c 调用 Complex 的成员函数？

3) 如果函数的返回值是类 A 的对象，则函数返回时，类 A 的复制构造函数被调用。换言之，作为函数返回值的对象是用复制构造函数初始化 的，而调用复制构造函数时的实参，就是 return 语句所返回的对象。例如下面的程序：

```
01.  #include<iostream>
02.  using namespace std;
03.  class A {
04.  public:
05.      int v;
06.      A(int n) { v = n; };
07.      A(const A & a) {
08.          v = a.v;
09.          cout << "Copy constructor called" << endl;
10.      }
11.  };
12.
13.  A Func() {
14.      A a(4);
15.      return a;
```

```
16.  }
17.
18.  int main() {
19.      cout << Func().v << endl;
20.      return 0;
21.  }
```

程序的输出结果是：
Copy constructor called
4

第19行调用了 Func 函数，其返回值是一个对象，该对象就是用复制构造函数初始化的， 而且调用复制构造函数时，实参就是第 16 行 return 语句所返回的 a。复制构造函数在第 9 行确实完成了复制的工作，所以第 19 行 Func 函数的返回值和第 14 行的 a 相等。

需要说明的是，有些编译器出于程序执行效率的考虑，编译的时候进行了优化，函数返回值对象就不用复制构造函数初始化了，这并不符合 C++ 的标准。上面的程序，用 Visual Studio 2010 编译后的输出结果如上所述，但是在 Dev C++ 4.9 中不会调用复制构造函数。把第 14 行的 a 变成全局变量，才会调用复制构造函数。对这一点，读者不必深究。