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 plain copy

```
1. int a = 100;
2. 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 plain copy

```
1. #include <iostream>
2. using namespace std;
3.
4. class CExample {
5. private:
6.       int a;
7. public:
8.       //构造函数
9.       CExample(int b)
10.      { a = b;}
11.
12.      //一般函数
13.      void Show ()
14.      {
15.        cout<<a<<endl;
16.      }
17. };
18.
19. int main()
20. {
21.      CExample A(100);
22.      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
23.       B.Show ();
24.      return 0;
25. }
```

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 plain copy

```
#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 plain copy

```
1.  class CExample
2.  {
3.  private:
4.   int a;
5.
6.  public:
7.   //构造函数
8.   CExample(int b)
9.   {
10.   a = b;
11.   cout<<"creat: "<<a<<endl;
12.  }
13.
14.  //拷贝构造
15.  CExample(const CExample& C)
16.  {
17.   a = C.a;
18.   cout<<"copy"<<endl;
19.  }
20.
21.  //析构函数
22.  ~CExample()
23.  {
24.   cout<< "delete: "<<a<<endl;
25.  }
26.
27.      void Show ()
28.  {
29.          cout<<a<<endl;
30.      }
31. };
32.
33. //全局函数，传入的是对象
34. void g_Fun(CExample C)
35. {
36.  cout<<"test"<<endl;
37. }
38.
39. 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 plain copy

```
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
```

```
27. CExample g_Fun()
28. {
29.  CExample temp(0);
30.  return temp;
31. }
32.
33. int main()
34. {
35.  g_Fun();
36.  return 0;
37. }
```

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 plain copy

```
1.  CExample A(100);
2.  CExample B = A;
3.  // 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 plain copy

```
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 plain copy

```
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 plain copy

```c
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 plain copy

```
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 plain copy

```
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 plain copy

```
// 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 plain copy

```
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 plain copy

```
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 plain copy

```
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 plain copy

```
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。

如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式：const 类型标识符 & 引用名=目标变量名;

例 1

```
int a ;

const int &ra=a;

ra=1; //错误

a=1; //正确
```

例 2

```
string foo( );

void bar(string & s);
```

那么下面的表达式将是非法的：

```
bar(foo( ));

bar("hello world");
```

原因在于 foo( )和"hello world"串都会产生一个临时对象，而在 C++中，这些临时对象都是 const 类型的。因此上面的表达式就是试图将一个 const 类型的对象转换为非 const 类型，这是非法的。

引用型参数应该在能被定义为 const 的情况下，尽量定义为 const 。

> 深拷贝和浅拷贝可以简单理解为：如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝，反之，没有重新分配资源，就是浅拷贝

1.什么是拷贝构造函数：

拷贝构造函数嘛，当然就是拷贝和构造了。（其实很多名字，只要静下心来想一想，就真的是顾名思义呀）拷贝又称复制，因此拷贝构造函数又称复制构造函数。百度百科上

是这样说的：拷贝构造函数，是一种特殊的构造函数，它由编译器调用来完成一些基于同一类的其他对象的构建及初始化。其唯一的参数（对象的引用）是不可变的（const 类型）。此函数经常用在函数调用时用户定义类型的值传递及返回。

2.拷贝构造函数的形式

复制代码代码如下:

```
Class X
{
public:
    X();
    X(const X&);//拷贝构造函数
}
```

2.1 为什么拷贝构造参数是引用类型？

其原因如下：当一个对象以传递值的方式传一个函数的时候，拷贝构造函数自动被调用来生成函数中的对象（符合拷贝构造函数调用的情况）。如果一个对象是被传入自己的拷贝构造函数，它的拷贝构造函数将会被调用来拷贝这个对象，这样复制才可以传入它自己的拷贝构造函数，这会导致无限循环直至栈溢出（Stack Overflow）。

3.拷贝构造函数调用的三种形式

3.1. 一个对象作为函数参数，以值传递的方式传入函数体；

3.2. 一个对象作为函数返回值，以值传递的方式从函数返回；

3.3. 一个对象用于给另外一个对象进行初始化（常称为复制初始化）。

总结：当某对象是按值传递时（无论是作为函数参数，还是作为函数返回值），编译器都会先建立一个此对象的临时拷贝，而在建立该临时拷贝时就会调用类的拷贝构造函数。

4.深拷贝和浅拷贝

如果在类中没有显式地声明一个拷贝构造函数，那么，编译器将会自动生成一个默认的拷贝构造函数，该构造函数完成对象之间的位拷贝。（位拷贝又称浅拷贝，后面将进行说明。）自定义拷贝构造函数是一种良好的编程风格，它可以阻止编译器形成默认的拷贝构造函数，提高源码效率。

在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如 A=B。这时，如果 B 中有一个成员变量指针已经申请了

内存，那 A 中的那个成员变量也指向同一块内存。这就出现了问题：当 B 把内存释放了（如：析构），这时 A 内的指针就是野指针了，出现运行错误。事实上这就要用到深拷贝了，要自定义拷贝构造函数。

深拷贝和浅拷贝可以简单理解为：如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝，反之，没有重新分配资源，就是浅拷贝。下面举个深拷贝的例子。

复制代码代码如下:

```
#include <iostream>
using namespace std;
class CA
{
public:
    CA(int b,char* cstr)
    {
        a=b;
         str=new char[b];
        strcpy(str,cstr);
    }
    CA(const CA& C)
    {
        a=C.a;
        str=new char[a]; //深拷贝
        if(str!=0)
           strcpy(str,C.str);
    }
    void Show()
    {
        cout<<str<<endl;
    }
    ~CA()
    {
        delete str;
    }
  private:
```

```
    int a;
    char *str;
};
int main()
{
    CA A(10,"Hello!");
    CA B=A;
    B.Show();
    return 0;
}
```

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。一定要注意类中是否存在指针成员。

5.拷贝构造函数与"="赋值运算符

例如：

复制代码代码如下:

```
class CExample
{};
int main()
{
CExample e1 = new CExample;
CExample e2 = e1;//调用拷贝构造函数
CExample e3(e1);//调用拷贝构造函数
CExample e4;
e4 = e1;//调用=赋值运算符
}
```

通常的原则是：①对于凡是包含动态分配成员或包含指针成员的类都应该提供拷贝构造函数；②在提供拷贝构造函数的同时，还应该考虑重载"="赋值操作符号。

```
Student s1("Jenny");
Student   s2=s1;
对象作为函数参数传递时，也要涉及对象的拷贝
void     fn(Student   fs)
{
```

```
        //...
}

int main()
{
  Student   ms;
  fn(ms);
}
```

函数 fn 的参数传递方式是传值，参数类型是 Student,调用时，实参 ms 传给了形参 fs,ms 在传递
的过程中是不会改变的,形参 fs 是 ms 的一个拷贝，这一切是在调用的开始完成的，也就是说，形参
fs 用 ms 的值进行构造。
这时候调用构造函数 Student(char*)就不合适，新的构造函数的参数应是 Student&,也就是：

```
Student (Student &s);
```

为什么 C++要用上面的拷贝构造函数，而他自己不会作像下面的事呢？

```
int   a=5;
int   b=a;
```

因为对象的类型多种多样，不像基本数据类型这么简单，有些对象还申请了系统资源，如 s 对象拥有了一个资源,用 s 的值创建一个 t 对象，如果仅仅只是二进制内存空间上的 s 拷贝，那意味着 t 也拥有这个资源
，由于资源归属权不清，将引起资源管理的混乱。

```
#include
using namespace   std;

class     Student
{
  public:
    Student (char *pName="no name",int ssId=0);
    Student (Student &s);
    ~Student ();

  protected :
```

```
    char name[40];
    int   id;
};
  Student:: Student (char *pName,int ssId)
{
      id=ssId;
      strcpy(name,pName);
      cout<<"Constructing   new   student"<<pname<<endl;
}
  Student::Student (Student &s)
  {
    cout <<"Constructing copy of "<<s.name<<endl;
    strcpy(name,"copy of ");
    strcat(name, s.name);
    id=s.id;

  }
  Student::~Student ()
{
     cout <<"Destructing "<<name<<endl;
}
void fn(Student s)
{
  cout<<"In   function   fn()\n";
}
int main()
{
  Student randy("Randy",1234);
  cout<<"Calling fn()\n";
  fn(randy);
  cout<<"Returned from   fn()\n";
}
```

如果拷贝构造函数中的参数不是一个引用，即形如 CClass(const CClass c_class)，那么就相当于采用了传值的方式(pass-by-value)，而传值的方式会调用该类的拷贝构造函数，从而造成无穷递归地调用拷贝构造函数。因此拷贝构造函数的参数必须是一个引用

在 C++中， 构造函数，拷贝构造函数，析构函数和赋值函数(赋值运算符重载)是最基本不过的需要掌握的知识。 但是如果我问你"拷贝构造函数的参数为什么必须使用引用类型？"这个问题， 你会怎么回答？ 或许你会回答为了减少一次内存拷贝？ 很惭愧的是，我的第一感觉也是这么回答。不过还好，我思索一下以后，发现这个答案是不对的。

原因：

如果拷贝构造函数中的参数不是一个引用，即形如 CClass(const CClass c_class)，那么就相当于采用了传值的方式(pass-by-value)，而传值的方式会调用该类的拷贝构造函数，从而造成无穷递归地调用拷贝构造函数。因此拷贝构造函数的参数必须是一个引用。

需要澄清的是，传指针其实也是传值，如果上面的拷贝构造函数写成 CClass(const CClass* c_class)，也是不行的。事实上，只有传引用不是传值外，其他所有的传递方式都是传值。

先从一个小例子开始：（自己测试一下自己看看这个程序的输出是什么？）

复制代码代码如下:

```
#include<iostream>
using namespace std;
class CExample
{
private:
 int m_nTest;
public:
 CExample(int x) : m_nTest(x)      //带参数构造函数
 {
  cout << "constructor with argument"<<endl;
 }
 // 拷贝构造函数，参数中的 const 不是严格必须的，但引用符号是必须的
 CExample(const CExample & ex)     //拷贝构造函数
 {
  m_nTest = ex.m_nTest;
  cout << "copy constructor"<<endl;
 }
 CExample& operator = (const CExample &ex)   //赋值函数(赋值运算符重载)
```

```
{
 cout << "assignment operator"<<endl;
 m_nTest = ex.m_nTest;
 return *this;
}
void myTestFunc(CExample ex)
{
}
};
int main(void)
{
 CExample aaa(2);
 CExample bbb(3);
 bbb = aaa;
 CExample ccc = aaa;
 bbb.myTestFunc(aaa);
 return 0;
}
```

果你能一眼看出就是这个结果的话， 恭喜你，可以站起来扭扭屁股，不用再往下看了。如果你的结果和输出结果有误差， 那拜托你谦虚的看完。

第一个输出： **constructor with argument // CExample aaa(2);**

如果你不理解的话， 找个人把你拖出去痛打一顿，然后嘴里还喊着"我是二师兄，我是二师兄....."

第二个输出：**constructor with argument // CExample bbb(3);**

分析同第一个

第三个输出： **assignment operator // bbb = aaa;**

第四个输出： **copy constructor // CExample ccc = aaa;**

这两个得放到一块说。 肯定会有人问为什么两个不一致。原因是， bbb 对象已经实例化了，不需要构造，此时只是将 aaa 赋值给 bbb，只会调用赋值函数，就这么简单，还不懂的话，撞墙去！ 但是 ccc 还没有实例化，因此调用的是拷贝构造函数，构造出 ccc，而不是赋值函数，还不懂的话，我撞墙去！！

第五个输出： **copy constructor // bbb.myTestFunc(aaa);**

实际上是 aaa 作为参数传递给 bbb.myTestFunc(CExample ex)， 即 CExample ex = aaa；和第四个一致的， 所以还是拷贝构造函数，而不是赋值函数， 如果仍然不懂，我的头刚才已经流血了，不要再让我撞了，你就自己使劲的再装一次吧。

通过这个例子， 我们来分析一下为什么拷贝构造函数的参数只能使用引用类型。

看第四个输出： **copy constructor // CExample ccc = aaa;**

构造 ccc，实质上是 ccc.CExample(aaa); 我们假如拷贝构造函数参数不是引用类型的话，那么将使得 ccc.CExample(aaa)变成 aaa 传值给 ccc.CExample(CExample ex)，即 CExample ex = aaa，因为 ex 没有被初始化， 所以 CExample ex = aaa 继续调用拷贝构造函数，接下来的是构造 ex，也就是 ex.CExample(aaa)，必然又会有 aaa 传给 CExample(CExample ex), 即 CExample ex = aaa;那么又会触发拷贝构造函数，就这下永远的递归下去。

所以绕了那么大的弯子，就是想说明拷贝构造函数的参数使用引用类型不是为了减少一次内存拷贝， 而是避免拷贝构造函数无限制的递归下去。

附带说明，在下面几种情况下会调用拷贝构造函数：

a、显式或隐式地用同类型的一个对象来初始化另外一个对象。如上例中，用对象 c 初始化 d；

b、作为实参(argument)传递给一个函数。如 CClass(const CClass c_class)中，就会调用 CClass 的拷贝构造函数；

c、在函数体内返回一个对象时，也会调用返回值类型的拷贝构造函数；

d、初始化序列容器中的元素时。比如 vector<string> svec(5)，string 的缺省构造函数和拷贝构造函数都会被调用；

e、用列表的方式初始化数组元素时。string a[] = {string("hello"), string("world")}; 会调用 string 的拷贝构造函数。

如果在没有显式声明构造函数的情况下，编译器都会为一个类合成一个缺省的构造函数。如果在一个类中声明了一个构造函数，那么就会阻止编译器为该类合成缺省的构造函数。和构造函数不同的是，即便定义了其他构造函数(但没有定义拷贝构造函数)，编译器总是会为我们合成一个拷贝构造函数。

另外函数的返回值是不是引用也有很大的区别，返回的不是引用的时候，只是一个简单的对象，此时需要调用拷贝构造函数，否则，如果是引用的话就不需要调用拷贝构造函数。

复制代码代码如下:

```
#include<iostream>
using namespace std;
class A
{
private:
```

```
 int m_nTest;
public:
 A()
 {
 }
 A(const A& other)    //构造函数重载
 {
  m_nTest = other.m_nTest;
  cout << "copy constructor"<<endl;
 }
 A & operator =(const A& other)
 {
  if(this != &other)
  {
   m_nTest = other.m_nTest;
   cout<<"Copy Assign"<<endl;
  }
  return *this;
 }
};
A fun(A &x)
{
 return x;     //返回的不是引用的时候，需要调用拷贝构造函数
}
int main(void)
{
 A test;
 fun(test);
 system("pause");
 return 0;
}
```

分享一道笔试题目，编译运行下图中的C++代码，结果是什么？（A）编译错误；（B）编译成功，运行时程序崩溃；（C）编译运行正常，输出10。请选择正确答案并分析原因。

复制代码代码如下:

```
class A
{
private:
 int value;
public:
 A(int n)
 {
  value = n;
 }
 A(A other)
 {
  value = other.value;
 }
 void Print()
 {
  cout<<value<<endl;
 }
};
int main(void)
{
 A a = 10;
 A b = a;
 b.Print();
 return 0;
}
```

答案：编译错误。在复制构造函数中传入的参数是 A 的一个实例。由于是传值，把形参拷贝到实参会调用复制构造函数。因此如果允许复制构造函数传值，那么会形成永无休止的递归并造成栈溢出。因此 C++的标准不允许复制构造函数传值参数，而必须是传引用或者常量引用。在 Visual Studio 和 GCC 中，都将编译出错。

一、C++中拷贝构造函数的定义：

有一个参数的类型是其类类型的构造函数是为拷贝构造函数。

如下所示：

```
?
```

```
X::X( const X& x);

Y::Y( const Y& y, int =0 );

//可以是多参数形式，但其第二个即后继参数都有一个默认值
```

二、拷贝构造函数的应用：

当一个类对象以另一个同类实体作为初值时，大部分情况下会调用拷贝构造函数。 一般是这三种具体情况：

1.显式地以一个类对象作为另一个类对象的初值，形如 X xx=x;

2.当类对象被作为参数交给函数时。

3.当函数返回一个类对象时。

后两种情形会产生一个临时对象。

三、C++中编译器何时合成拷贝构造函数

并不是所有未定义有拷贝构造函数的类编译器都会为其合成拷贝构造函数，编译器只有在必要的时候才会为其合成拷贝构造函数。所谓必要的时刻是指编译器在普通手段无法完成解决"当一个类对象以另一个同类实体作为初值"时，才会合成拷贝构造函数。也就是说，当常规手段能解决问题的时候，就没必要动用非常规手段。

如果一个类没有定义拷贝构造函数，通常按照"成员逐一初始化(Default Memberwise Initialization)"的手法来解决"一个类对象以另一个同类实体作为 初值"——也就是说把内建或派生的数据成员从某一个对象拷贝到另一个对象身上，如果数据成员是一个对象，则递归使用"成员逐一初始化(Default Memberwise Initialization)"的手法。

成员逐一初始化(Default Memberwise Initialization)具体的实现方式则是位 逐次拷贝（Bitwise copy semantics）1。也就是说在能使用这种常规方式 来解决"一个类对象以另一个同类实体作为初值"的时候，编译器是不需要合成拷 贝构造函数的。但有些时候常规武器不那么管用，我们就得祭出非常规武器了 ——拷贝构造函数。有以下几种情况之一，位逐次拷贝将不能胜任或者不适合来完成"一个类对象以另一个同类实体作为初值"的工作。此时，如果类没有定义拷贝 构造函数，那么编译器将必须为类合成一个拷贝构造函数。

当类内含一个成员对象，而后者的类声明有一个拷贝构造函数时（不论是设 计者定义的还是编译器合成的）。

当类继承自一个声明有拷贝构造函数的类时（同样，不论这个拷贝构造函数 是被显示声明还是由编译器合成的）。

四、类中声明有虚函数

当类的派生串链中包含有一个或多个虚基类。

对于前两种情况，不论是基类还是对象成员，既然后者声明有拷贝构造函数时， 就表明其类的设计者或者编译器希望以其声明的拷贝构造函数来完成"一个类对象 以另一个同类实

体作为初值”的工作，而设计者或编译器这样做——声明拷贝构造函 数，总有它们的理由，而通常最直接的原因莫过于因为他们想要做一些额外的工 作或“位逐次拷贝”无法胜任。

对于有虚函数的类，如果两个对象的类型相同那么位逐次拷贝其实是可以胜任的。但问题将出现在，如果基类由其继承类进行初始化时，此时若按照位逐次拷贝来完成这个工作，那么基类的 vptr 将指向其继承类的虚函数表，这将导致无法预料的后果——调用一个错误的虚函数实体是无法避免的，轻则带来程序崩溃，更糟糕的问题可能是这个错误被隐藏了。所以对于有虚函数的类编译器将会明确的使被 初始化的对象的 vptr 指向正确的虚函数表。因此有虚函数的类没有声明拷贝构造 函数，编译将为之合成一个，来完成上述工作，以及初始化各数据成员，声明有 拷贝构造函数的话也会被插入完成上述工作的代码。

对于继承串链中有虚基类的情况，问题同样出现在继承类向基类提供初值的情况， 此时位逐次拷贝有可能破坏对象中虚基类子对象的位置。