深拷贝和浅拷贝：

CA(const CA& 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别的构造函数，函数的名称必须和类名称一致，他的唯一的一个参数是本类型的一个引用变量， 该参数是 const 类型，不可变的。例如：类 X 的拷贝构造函数的形式为 X(X& x)。
当用一个已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。也就是说，当类的对象需要拷贝时，拷贝构造函数将会被调用。以下情况都会调用拷贝构造函数：

- 一个对象以值传递的方式传入函数体
- 一个对象以值传递的方式从函数返回
- 一个对象需要通过另外一个对象进行初始化。

假如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，他能够阻止编译器形成默认的拷贝构造函数，提高源码效率。
浅拷贝和深拷贝
在某些状况下，类内成员变量需要动态开辟堆内存，假如实行位拷贝，也就是把对象里的值完全复制给另一个对象，如 A=B。这时，假如 B 中有一个成员变量指针已申请了内存，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。

深拷贝和浅拷贝能够简单理解为：假如一个类拥有资源，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

浅 拷贝就是成员数据之间的一一赋值：把值赋给一一赋给要拷贝的值。但是可能会有这样的情况:对象还包含资源,这里的资源可以值堆资源,或者一个文件。。当 值拷贝的时候，两个对象就有用共同的资源，同时对资源可以访问，这样就会出问题。深拷贝就是用来解决这样的问题的，它把资源也赋值一次，使对象拥有不同的 资源，但资源的内容是一样的。对于堆资源来说，就是在开辟一片堆内存，把原来的内容拷贝。

如果你拷贝的对象中引用了某个外部的内容（比如分配在堆上的数据），那么在拷贝这个对象的时候，让新旧两个对象指向同一个外部的内容，就是浅拷贝；如果在拷贝这个对象的时候为新对象制作了外部对象的独立拷贝，就是深拷贝

引用和指针的语义是相似的，引用是不可改变的指针，指针是可以改变的引用。其实都是实现了引用语义。
深拷贝和浅拷贝的区别是在对象状态中包含其它对象的引用的时候，当拷贝一个

对象时，如果需要拷贝这个对象引用的对象，则是深拷贝，否则是浅拷贝。

COW 语义是“深拷贝”与“推迟计算”的组合，仍然是深拷贝，而非浅拷贝，因为拷贝之后的两个对象的数据在逻辑上是不相关的，只是内容相同。

无论深浅，都是需要的。当深拷贝发生时，通常表明存在着一个“聚合关系”，而浅拷贝发生时，通常表明存在着一个“相识关系”。
举个简单的例子：
当你实现一个 Composite Pattern，你通常都会实现一个深拷贝(如果需要拷贝的话)，很少有要求同的 Composite 共享 Leaf 的；
而当你实现一个 Observer Pattern 时，如果你需要拷贝 Observer,你大概不会去拷贝 Subject，这时就要实现个浅拷贝。
是深拷贝还是浅拷贝，并不是取决于时间效率、空间效率或是语言等等，而是取决于哪一个是逻辑上正确的

//--

在学习这一章内容前我们已经学习过了类的构造函数和析构函数的相关知识，对于普通类型的对象来说，他们之间的复制是很简单的，例如：

```
int a = 10;

int b =a;
```

自己定义的类的对象同样是对象，谁也不能阻止我们用以下的方式进行复制,例如：

```
#include <iostream>

usingnamespacestd;



classTest

{

public:

Test(inttemp)

{
```

```
p1=temp;

}

protected:

intp1;



};



voidmain()

{

Test a(99);

Test b=a;

}
```

普 通对象和类对象同为对象，他们之间的特性有相似之处也有不同之处，类对象内部存在成员变量，而普通对象是没有的，当同样的复制方法发生在不同的对象上的 时候，那么系统对他们进行的操作也是不一样的，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，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！

代码如下：

```
#include <iostream>

usingnamespacestd;



```

```
classTest

{

public:

Test(inttemp)

{

p1=temp;

}

Test(Test &c_t)//这里就是自定义的拷贝构造函数

{

cout<<"进入 copy 构造函数"<p1=c_t.p1;//这句如果去掉就不能完成复制工作了,此句复制过程的核心语句

}

public:

intp1;

};



voidmain()

{

Test a(99);

Test b=a;

cout<cin.get();

}
```

上面代码中的 Test(Test &c_t)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变量,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 Test(Test &c_t)拷贝构造函数内的 p1=c_t.p1;语句完成的。如果取掉这句代码，那么 b 对象的 p1 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，那么我们没有意义要去自定义拷贝构造函数呀，对，就普通情况 而言这的确是没有必要的，但在某写状况下，类体内的成员是需要开辟动态开辟堆内存的,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，已经开辟的一端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。

为了更详细的说明问题，请看如下的代码。

```
#include <iostream>

usingnamespacestd;



classInternet

{

public:

Internet(char*name,char*address)

{

cout<<"载入构造函数"<strcpy(Internet::name,name);

strcpy(Internet::address,address);

cname=newchar[strlen(name)+1];
```

```
if(cname!=NULL)

{

strcpy(Internet::cname,name);

}

}

Internet(Internet &temp)

{

cout<<"载入 COPY 构造函数
"<strcpy(Internet::name,temp.name);

strcpy(Internet::address,temp.address);

cname=newchar[strlen(name)+1];//这里注意,深拷贝的体现!

if(cname!=NULL)

{

strcpy(Internet::cname,name);

}

}

~Internet()

{

cout<<"载入析构函数!";

delete[] cname;

cin.get();

}
```

```
voidshow();

protected:

charname[20];

charaddress[30];

char*cname;

};

voidInternet::show()

{

cout<}

voidtest(Internet ts)

{

cout<<"载入 test 函数"<}

voidmain()

{

Internet a("中国软件开发实验室
","www.cndev-lab.com");

Internet b =a;

b.show();

test(b);

}
```

上面代码就演示了深拷贝的问题,对对象 b 的 cname 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,最后我必须提一下，对于上面的程序我的解释并不多，就是希望读者本身运行程序观察变化，进而深刻理解。

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!

以前我们的教程中讨论过函数返回对象产生临时变量的问题,接下来我们来看一下在函数中返回自定义类型对象是否也遵循此规则产生临时对象!

先运行下列代码:

```
#include <iostream>

usingnamespacestd;



classInternet

{

public:

Internet()

{



};

Internet(char*name,char*address)

{

cout<<"载入构造函数"<strcpy(Internet::name,name);

strcpy(Internet::address,address);

}
```

```
Internet(Internet &temp)

{

cout<<"载入 COPY 构造函数
"<strcpy(Internet::name,temp.name);

strcpy(Internet::address,temp.address);

cin.get();

}

~Internet()

{

cout<<"载入析构函数!";

cin.get();

}

protected:

charname[20];

charaddress[20];

};

Internet tp()

{

Internet b("中国软件开发实验室
","www.cndev-lab.com");

returnb;

}

voidmain()
```

```
{

Internet a;

a=tp();

}
```

从上面的代码运行结果可以看出，程序一共载入过析构函数三次，证明了由函数返回自定义类型对象同样会产生临时变量，事实上对象 a 得到的就是这个临时 Internet 类类型对象 temp 的值。

这一下节的内容我们来说一下无名对象。

利用无名对象初始化对象系统不会不调用拷贝构造函数。

那么什么又是无名对象呢？

很简单，如果在上面程序的 main 函数中有：

```
Internet  ("中国软件开发实验室
","www.cndev-lab.com");
```

这样的一句语句就会产生一个无名对象，无名对象会调用构造函数但利用无名对象初始化对象系统不会不调用拷贝构造函数！

下面三段代码是很见到的三种利用无名对象初始化对象的例子。

```
#include <iostream>

usingnamespacestd;



classInternet

{

public:

Internet(char*name,char*address)

{
```

```
cout<<"载入构造函数"<strcpy(Internet::name,name);

}

Internet(Internet &temp)

{

cout<<"载入 COPY 构造函数
"<strcpy(Internet::name,temp.name);

cin.get();

}

~Internet()

{

cout<<"载入析构函数!";

cin.get();

}

public:

charname[20];

charaddress[20];

};



voidmain()

{

Internet a=Internet("中国软件开发实验室
","www.cndev-lab.com");

cout<cin.get();
```

```
}
```

上面代码的运行结果有点“出人意料”，从思维逻辑上说，当无名对象创建了后，是应该调用自定义拷贝构造函数，或者是默认拷贝构造函数来完成复制过程的，但事实上系统并没有这么做，因为无名对象使用过后在整个程序中就失去了作用，对于这种情况 c++会把代码看成是：

Internet a("中国软件开发实验室","www.cndev-lab.com");

省略了创建无名对象这一过程，所以说不会调用拷贝构造函数。

最后让我们来看看引用无名对象的情况。

```
#include <iostream>

usingnamespacestd;



classInternet

{

public:

Internet(char*name,char*address)

{

cout<<"载入构造函数"<strcpy(Internet::name,name);

}

Internet(Internet &temp)

{

cout<<"载入 COPY 构造函数
"<strcpy(Internet::name,temp.name);

cin.get();

}
```

```
~Internet()

{

cout<<"载入析构函数!";

}

public:

charname[20];

charaddress[20];

};



voidmain()

{

Internet &a=Internet("中国软件开发实验室
","www.cndev-lab.com");

cout<cin.get();

}
```

引用本身是对象的别名，和复制并没有关系，所以不会调用拷贝构造函数，但要注意的是，在 c++看来:

```
Internet &a=Internet("中国软件开发实验室
","www.cndev-lab.com");
```

是等价与:

```
Internet a("中国软件开发实验室
","www.cndev-lab.com");
```

的,注意观察调用析构函数的位置（这种情况是在 main()外调用，而无名对象本身是在 main()内析构的）

拷贝构造函数和赋值构造函数的异同

由于并非所有的对象都会使用拷贝构造函数和赋值函数，程序员可能对这两个函数
有些轻视。请先记住以下的警告，在阅读正文时就会多心：
?? 如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”
的方式自动生成缺省的函数。倘若类中含有指针变量，那么这两个缺省的函数就隐
含了错误。以类 String 的两个对象 a,b 为例，假设 a.m_data 的内容为“hello”，
b.m_data 的内容为“world”。
现将 a 赋给 b，缺省赋值函数的“位拷贝”意味着执行 b.m_data = a.m_data。
这将造成三个错误：一是 b.m_data 原有的内存没被释放，造成内存泄露；二是
b.m_data 和 a.m_data 指向同一块内存，a 或 b 任何一方变动都会影响另一方；三
是在对象被析构时，m_data 被释放了两次。
?? 拷贝构造函数和赋值函数非常容易混淆，常导致错写、错用。拷贝构造函数是在对
象被创建时调用的，而赋值函数只能被已经存在了的对象调用。以下程序中，第三
个语句和第四个语句很相似，你分得清楚哪个调用了拷贝构造函数，哪个调用了赋
值函数吗？

```
String a(“hello”);
String b(“world”);
String c = a; // 调用了拷贝构造函数，最好写成 c(a);
c = b; // 调用了赋值函数
```

本例中第三个语句的风格较差，宜改写成 String c(a) 以区别于第四个语句。
类 String 的拷贝构造函数与赋值函数

```
// 拷贝构造函数
String::String(const String &other)
{
// 允许操作 other 的私有成员 m_data
int length = strlen(other.m_data);
m_data = new char[length+1];
strcpy(m_data, other.m_data);
}
// 赋值函数
String & String::operate =(const String &other)
{
// (1) 检查自赋值
```

```
if(this == &other)
return *this;
// (2) 释放原有的内存资源
delete [] m_data;
// （3）分配新的内存资源，并复制内容
int length = strlen(other.m_data);
m_data = new char[length+1];
strcpy(m_data, other.m_data);
// （4）返回本对象的引用
return *this;
}
```

类 String 拷贝构造函数与普通构造函数的区别是：在函数入口处无需与 NULL 进行比较，这是因为“引用”不可能是 NULL，而“指针”可以为 NULL。

类 String 的赋值函数比构造函数复杂得多，分四步实现：

（1）第一步，检查自赋值。你可能会认为多此一举，难道有人会愚蠢到写出 a = a 这样的自赋值语句！的确不会。但是间接的自赋值仍有可能出现，例如

```
// 内容自赋值
b = a;
…
c = b;
…
a = c;
// 地址自赋值
b = &a;
…
a = *b;
```

也许有人会说：“即使出现自赋值，我也可以不理睬，大不了化点时间让对象复制自己而已，反正不会出错！”

他真的说错了。看看第二步的 delete，自杀后还能复制自己吗？所以，如果发现自赋值，应该马上终止函数。注意不要将检查自赋值的 if 语句

```
if(this == &other)
```

错写成为

```
if( *this == other)
```

（2）第二步，用 delete 释放原有的内存资源。如果现在不释放，以后就没机会了，将造成内存泄露。

（3）第三步，分配新的内存资源，并复制字符串。注意函数 strlen 返回的是有效字符串长度，不包含结束符‘\0’。函数 strcpy 则连‘\0’一起复制。

（4）第四步，返回本对象的引用，目的是为了实现象 a = b = c 这样的链

式表达。注

意不要将 return *this 错写成 return this 。那么能否写成 return other 呢？效果

不是一样吗？

不可以！因为我们不知道参数 other 的生命期。有可能 other 是个临时对象，在赋

值结束后它马上消失，那么 return other 返回的将是垃圾。

偷懒的办法处理拷贝构造函数与赋值函数

如果我们实在不想编写拷贝构造函数和赋值函数，又不允许别人使用编译器生成的

缺省函数，怎么办？

偷懒的办法是：只需将拷贝构造函数和赋值函数声明为私有函数，不用编写代码。

例如：

```
class A
{ …
private:
A(const A &a); // 私有的拷贝构造函数
A & operate =(const A &a); // 私有的赋值函数
};
```

如果有人试图编写如下程序：

```
A b(a); // 调用了私有的拷贝构造函数
b = a; // 调用了私有的赋值函数
```

编译器将指出错误，因为外界不可以操作 A 的私有函数。

一、

拷贝构造，是一个的对象来初始化一边内存区域，这边内存区域就是你的新对象的内存区域赋值运算，对于一个已经被初始化的对象来进行 operator=操作

```
class     A;
  A   a;
  A   b=a;      //拷贝构造函数调用
  //或
  A   b(a);      //拷贝构造函数调用
  ////////////////////////////////
  A   a;
  A   b;
  b =a;      //赋值运算符调用
```

你只需要记住，在 C++语言里，

```
  String     s2(s1);
  String     s3     =     s1;
```

只是语法形式的不同，意义是一样的，都是定义加初始化，都调用拷贝构造函数。

二、
一般来说是在数据成员包含指针对象的时候,应付两种不同的处理需求的 一种是复制指针对象,一种是引用指针对象 copy 大多数情况下是复制,=则是引用对象的
例子:

```
class     A
{
                int     nLen;
                char     *     pData;
}
```

显然

```
A     a,     b;
```

a=b 的时候,对于 pData 数据存在两种需求
第一种 copy

```
        a.pData     =     new     char     [nLen];
        memcpy(a.pData,     b.pData,     nLen);
```

另外一种(引用方式):

```
        a.pData     =     b.pData
```

通过对比就可以看到,他们是不同的
往往把第一种用 copy 使用,第二种用=实现
你只要记住拷贝构造函数是用于类中指针,对象间的 COPY

三、
和拷贝构造函数的实现不一样
拷贝构造函数首先是一个构造函数,它调用的时候产生一个对象,是通过参数传进来的那个对象来初始化,产生的对象。
operator=();是把一个对象赋值给一个原有的对象,所以如果原来的对象中有内存分配要先把内存释放掉,而且还要检查一下两个对象是不是同一个对象,如果是的话就不做任何操作。

还要注意的是拷贝构造函数是构造函数,不返回值

而赋值函数需要返回一个对象自身的引用,以便赋值之后的操作

你肯定知道这个:

```
int     a,     b;

b     =     7;

Func(     a     =     b     );     //     把 i 赋值后传给函数
Func(     int     )
```

同理:

```
CMyClass     obj1,     obj2;
```

```
obj1.Initialize();

Func2( obj1 = obj2 ); //如果没有返回引用，是不能把值传给 Func2 的
```

注:

```
CMyClass & CMyClass:: operator = ( CMyClass & other )
{
    if( this == &other )
        return *this;
    // 赋值操作...
    return *this
}
```