介绍下C++的类继承？，，，派生类的内存分布，； 。。C++继承方式，4种，包含虚继承，追问，protected和public继承区别，.B C虚继承A，D public继承 B C ，有A *a = new D，a->fun(),fun是虚函数，并且B C都重写了，怎么保证a调用的是B重写的虚函数。；5、，13、继承、虚继承14、钻石继承问题15、同名覆盖问题16、24、class A :class B{},A是私有继承还是？ 私有继承是做什么用的？菱形继承的虚函数的开销说一下，，class怎么防止继承，只讲了私有构造，忘了final关键字；。

面向对象

三大特性：封装,继承,多态

1、封装：封装是实现面向对象程序设计的第一步，封装就是将数据或函数等集合在一个个的单元中（我们称之为类）。封装的意义在于保护或者防止代码（数据）被我们无意中破坏，把它的一部分属性和功能对外界屏蔽。
2、继承：继承主要实现重用代码，节省开发时间。子类可以继承父类的一些东西。
3、多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。分为编译时多态和运行时多态。

C和C++的主要区别

1、C语言属于面向过程语言，通过函数来实现程序功能。而C++是面向对象语言，主要通过类的形式来实现程序功能。
2、使用C++编写的面向对象应用程序比C语言编写的程序更容易维护、扩展性更强。
3、C++多用于开发上层应用软件，而C语言代码体积小、执行效率高，多编写系统软件和嵌入式开发。

C不是C++的子集：
从实用角度讲，C++属于C语言的一个超集，基本上兼容ANSI C。但是从编译角度上讲，C语言的有些特性在C++中并不支持。C特有的特性：基本数据类型Bool_

面向对象与面向过程分别是什么？

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了；以过程为中心的编程思想，以算法进行驱动。

面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。以对象为中心，以消息进行驱动。

面向过程与面向对象的优缺点？

面向过程

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源，比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护、易复用、易扩展。

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护。

缺点：性能比面向过程低。

struct和class的区别：

1.只有结构体存在于C语言，结构体和类都存在于C++。

2.在C中结构体只涉及到数据结构，在C++中结构体和类体现了数据结构和算法的结合。也就是说在C++中，结构体还可以包含函数，而C语言中不可以。

3.在C语言中结构体没有成员函数，可以被声明为变量、指针和数组等。在C++中由于对结构体进行了扩充，获取了很多功能，当然还可以包含函数。

4.访问机制，结构体默认访问的机制是public，而类默认访问机制是private。

5.从class继承默认是private继承，而从struct继承默认是public继承。

6.C++的结构体声明不必有struct关键字，而C语言的结构体声明必须带有关键字（使用typedef别名定义除外）。

7.结构体是值类型，而类是引用类型。

8.结构体和类同样可以能够继承和实现多态。

9.在C++中，结构体可以继承类，类同样也可以继承结构体，继承过程中注意访问权限。

10.“class”这个关键字还用于定义模板参数，就像“typename”。但关键字“struct”不用于定义模板参数。这一点在Stanley B.Lippman写的Inside the C++ Object Model有过说明。

11.struct更适合看成是一个数据结构的实现体，class更适合看成是一个对象的实现体。

12.实例化的类存储在内存的堆内，而结构存储在栈内，结构的执行效率相对较高。

13.结构体没有析构函数。

哪种使用的情况下适合结构体还是类？

1.在表示诸如点、矩形等主要用来存储数据的轻量级对象时，首选struct。

2.在表示数据量大、逻辑复杂的大对象时，首选class。

3.在表现抽象和多级别的对象层次时，class是最佳选择。

多态

多态性可以简单地概括为“一个接口，多种方法”，程序在运行时才决定调用的函数，它是面向对象编程领域的核心概念。
C++多态性是通过虚函数来实现的，虚函数允许子类重新定义成员函数，而子类重新定义父类的做法称为覆盖(override)，或者称为重写。
多态可分为**静态多态和动态多态**。静态多态是指在编译期间就可以确定函数的调用地址，并生产代码，这就是静态的，也就是说地址是早早绑定的，静态多态也往往被叫做静态联编。 动态多态则是指函数调用的地址不能在编译器期间确定，必须需要在运行时才确定，这就属于晚绑定，动态多态也往往被叫做动态联编。 静态多态往往通过函数重载（运算符重载）和模版（泛型编程）来实现。

动态绑定

当使用基类的引用或指针调用虚成员函数时会执行动态绑定。动态绑定直到运行的时候才知道到底调用哪个版本的虚函数，所以必为每一个虚函数都提供定义，而不管它是否被用到，这是因为连编译器都无法确定到底会使用哪个虚函数。被调用的函数是与绑定到指针或引用上的对象的动态类型相匹配的那一个。

重载、覆盖（重写）、隐藏（重定义）

重载

1.在同一个作用域下，函数名相同，函数的参数不同（参数不同指参数的类型或参数的个数不相同）
2.不能根据返回值判断两个函数是否构成重载。
3.当函数构成重载后，调用该函数时，编译器会根据函数的参数选择合适的函数进行调用。

重写（覆盖）

在不同的作用域下（一个在父类，一个在子类），函数的函数名、参数、返回值完全相同，父类必须含有virtual关键字（协变除外）。子类重新定义父类中有相同名称和参数的虚函数。
重写需要注意：
1 被重写的函数不能是static的。必须是virtual的
2 重写函数必须有相同的类型，名称和参数列表
3 重写函数的访问修饰符可以不同。尽管virtual是private的，派生类中重写改写为public,protected也是可以的

重定义（隐藏）

1.在不同的作用域下（这里不同的作用域指一个在子类，一个在父类 ），函数名相同的两个函数构成重定义。（不是虚函数）
2.当两个函数构成重定义时，父类的同名函数会被隐藏，当用子类的对象调用同名的函数时，如果不指定类作用符，就只会调用子类的同名函数。
3.如果想要调用父类的同名函数，就必须指定父类的域作用符。
注意：当父类和子类的成员变量名相同时，也会构成隐藏。

类

定义

```
class 类名
{
private:
    数据成员或成员函数
protected:
    数据成员或成员函数
public:
    数据成员或成员函数
};
```

访问权限

(1)public（公有类型）：表示这个成员可以被该类对象处在同一作用域内的任何函数使用。一般将成员函数声明为公有的访问控制。

(2)private（私有类型）：表示这个成员能被它所在的类中的成员函数&该类的友元函数使用。

(3)protected（保护类型）：表示这个成员只能被它所在类&从该类派生的子类的成员函数&友元函数使用。

类的构造函数

每个类都分别定义了它的对象被初始化的方式，类通过一个或几个特殊的成员函数来控制其对象的初始化过程，这些函数叫构造函数。构造函数的任务是初始化类对象的数据成员，无论何时只要类的对象被创建，就会执行构造函数。

1、构造函数的名字和类名相同、无返回类型、有一个（可能为空的）参数列表和一个（可能为空的）函数体。

2、类可以包含多个构造函数，和其他重载函数差不多，不同的构造函数之间必须在参数数量或参数类型上有所区别。

3、不同于其他成员函数，构造函数不能被声明成const的。当我们创建类的一个const对象时，直到构造函数完成初始化过程，对象才能真正取得其“常亮”属性。因此，构造函数在const对象的构造过程中可以向其写值。

4、与其他成员函数相同的是，构造函数在类外定义时也需要明确指出是哪个类。

5、通常情况下，我们将构造函数声明为public的，可以供外部调用。然而有时候我们会将构造函数声明为private或protected的：（1）如果类的作者不希望用户直接构造一个类对象，着只是希望用户构造这

个类的子类，那么就可以将构造函数声明为protected，而将该类的子类声明为public。（2）如果将构造函数声明为private，那只有这个类的成员函数才能构造这个类的对象。

委托构造函数

在一个类中重载多个构造函数时，这些函数只是形参不同，初始化列表不同，而初始化算法和函数体都是相同的。这个时候，为了避免重复，C++11新标准提供了委托构造函数。更重要的是，可以保持代码的一致性，如果以后要修改构造函数的代码，只需要在一处修改即可。

```
class X {
        int a;
        // 实现一个初始化函数
        validate(int x) {
            if (0<x && x<=max) a=x; else throw bad_X(x);
        }
    public:
        // 三个构造函数都调用validate()，完成初始化工作
        X(int x) { validate(x); }
        X() { validate(42); }
        X(string s) {
            int x = lexical_cast<int>(s); validate(x);
        }
        // …
};
```

这样的实现方式重复罗嗦，并且容易出错。并且，这两种方式的可维护性都很差。所以，在C++0x中，我们可以在定义一个构造函数时调用另外一个构造函数：

```
class X {
        int a;
    public:
        X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
        // 构造函数X()调用构造函数X(int x)
        X() :X{42} { }
        // 构造函数X(string s)调用构造函数X(int x)
        X(string s) :X{lexical_cast<int>(s)} { }
        // …
};
```

拷贝构造函数

如果一个构造函数的第一个参数是自身类型的引用，且任何额外参数都有默认值，则此构造函数是拷贝构造函数。
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。

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

拷贝构造函数的调用时机

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;
```

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。
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：

```
void Function(const Complex & c)
{
    ...
}
```

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：

```
#include<iostream>
using namespace std;
class A {
public:
    int v;
    A(int n) { v = n; };
    A(const A & a) {
        v = a.v;
        cout << "Copy constructor called" << endl;
    }
};
A Func() {
    A a(4);
    return a;
}
int main() {
    cout << Func().v << endl;
    return 0;
}
```

程序的输出结果是：
Copy constructor called
4

拷贝构造函数的作用

作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实例。
上面说的三个调用时机，如果后两种不用拷贝构造函数的话，会导致一个指针指向已经删除的内存空间。对于第一种情况，初始化和赋值的不同含义是拷贝构造函数调用的原因，

重写拷贝构造函数的意义

因为如果不写拷贝构造函数，系统就只会调用默认构造函数，然而默认构造函数是一种浅拷贝。相当于只对指针进行了拷贝（位拷贝），而有些时候我们却需要拷贝整个构造函数包括指向的内存，这种拷贝被称为深拷贝（值拷贝）。

所以为了达成深拷贝的目的，自己手写拷贝构造函数是非常必要的。

深拷贝和浅拷贝

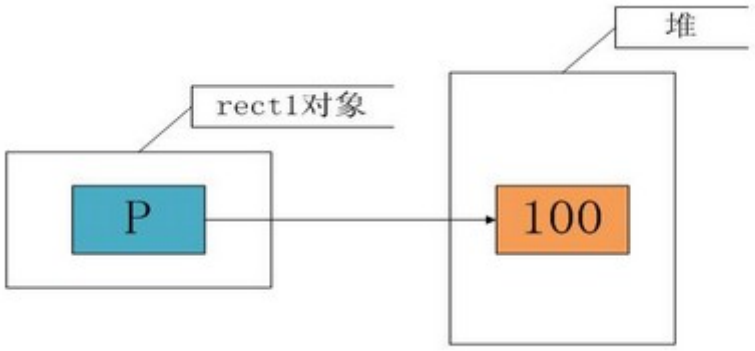
浅拷贝

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：

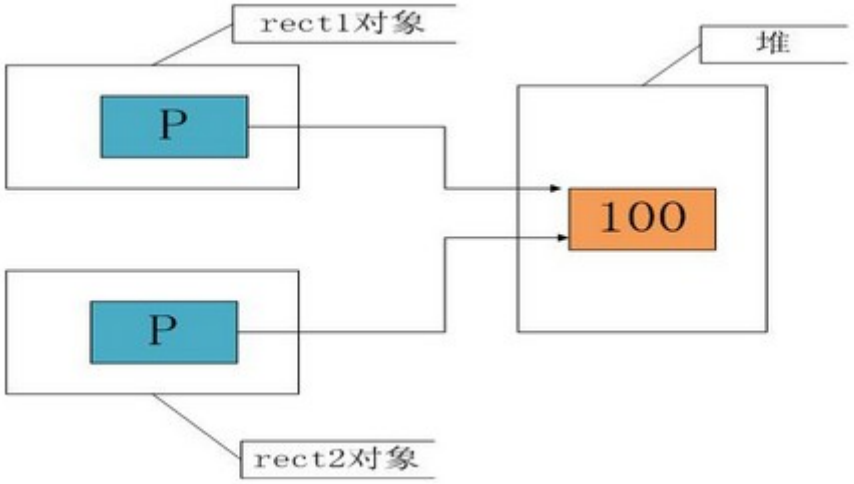
```
class Rect
{
public:
   Rect()
   {
       p=new int(100);
   }

   ~Rect()
   {
       assert(p!=NULL);
       delete p;
   }
private:
   int width;
   int height;
   int *p;
};
int main()
{
   Rect rect1;
   Rect rect2(rect1);
   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：



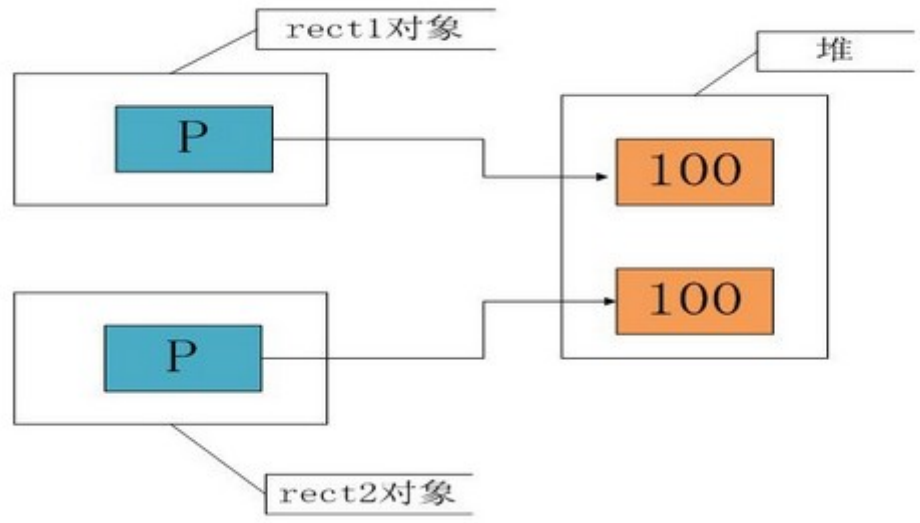
当然，这不是我们所期望的结果，在销毁对象时，两个对象的析构函数将对同一个内存空间释放两次，这就是错误出现的原因。我们需要的不是两个p有相同的值，而是两个p指向的空间有相同的值，解决办法就是使用“深拷贝”。

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()
  {
   p=new int(100);
  }

  Rect(const Rect& r)
  {
      width=r.width;
      height=r.height;
      p=new int(100);
      *p=*(r.p);
  }

  ~Rect()
  {
   assert(p!=NULL);
      delete p;
  }
private:
  int width;
  int height;
  int *p;
};
int main()
{
  Rect rect1;
  Rect rect2(rect1);
  return 0;
}
```

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”。

简而言之，当数据成员中有**指针**时，必须要用深拷贝。

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。

为什么参数为引用类型

简单的回答是为了防止递归引用。具体一些可以这么讲：当 一个对象需要以值方式传递时，编译器会生成代码调用它的拷贝构造函数以生成一个复本。如果类A的拷贝构造函数是以值方式传递一个类A对象作为参数的话，当需要调用类A的拷贝构造函数时，需要以值方式传进一个A的对象作为实参； 而以值方式传递需要调用类A的拷贝构造函数；结果就是调用类A的拷贝构造函数导致又一次调用类A的拷贝构造函数，这就是一个无限递归。

参数传递过程到底发生了什么？

将地址传递和值传递统一起来，归根结底还是传递的是"值"(地址也是值，只不过通过它可以找到另一个值)！
i)值传递:
对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量)；
对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象)；如void foo(class_type obj_local){}, 如果调用foo(obj); 首先class_type obj_local(obj) ,这样就定义了局部变量obj_local供函数内部使用
ii)引用传递:
无论对内置类型还是类类型，传递引用或指针最终都是传递的地址值！而地址总是指针类型(属于简单类型), 显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型).

析构函数

构造函数用于创建对象，而析构函数是用来撤销对象。
析构函数往往用来做“清理善后” 的工作（例如在建立对象时用new开辟了一片内存空间，应在退出前在析构函数中用delete释放）。

虚析构函数

虚析构函数可以认为是特殊的析构函数，主要作用在继承关系中。

若B是A的子类： A *a=new B;

delete a;

如果A的析构函数是non-vartual，则只会调用A的析构函数，这样B的资源没有释放，就会有内存泄露；

如果A的析构函数是vartual，则只会先调用A的析构函数，再调用B的析构函数。

多态和虚函数

虚函数

只需要在函数声明前面增加 virtual 关键字。虚函数是多态性的基础，其调用的方式是动态联编（程序运行时才决定调用基类的还是子类）。
虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数，达到多态的目的。

为什么有的类的析构函数需要定义成虚函数

C++中基类采用virtual虚析构函数是为了防止内存泄漏。具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

纯虚函数

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加 =0:
引入原因
1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。
2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

抽象类的作用是作为一个类族的共同基类，或者说，为一个类族提供一个公共接口。

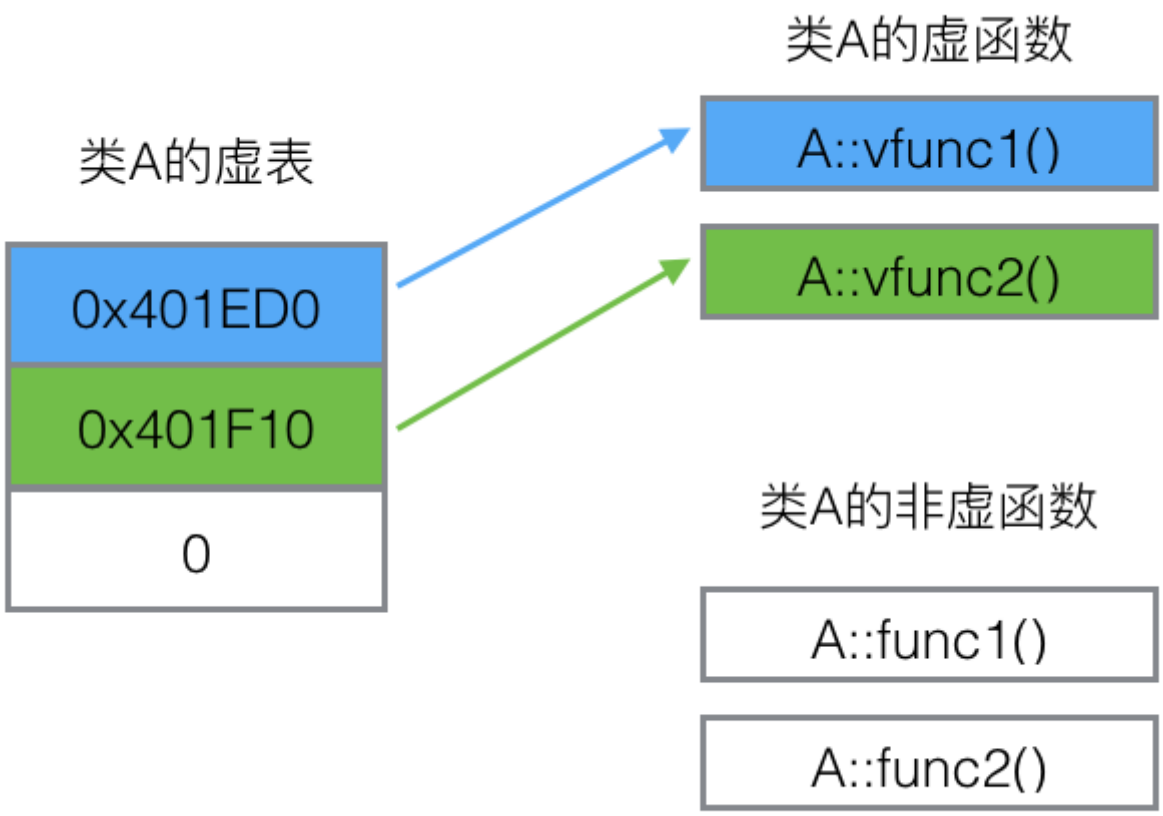
虚函数表

每一个包含了虚函数的类都有一个虚表。

1. 对于一个class，产生一堆指向virtual functions的指针，这些指针被统一放在一个表格中。这个表格被称为虚函数表，英文又称做virtual table（vtbl）。
2. 每一个对象中都添加一个指针，指向相关的virtual table。通常这个指针被称作虚函数表指针（vptr）。出于效率的考虑，该指针通常放在对象实例最前面的位置（第一个slot处）。每一个class所关联的type_info信息也由virtual table指出（通常放在表格的最前面）。

类A包含虚函数vfunc1，vfunc2，由于类A包含虚函数，故类A拥有一个虚表。

```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};
```

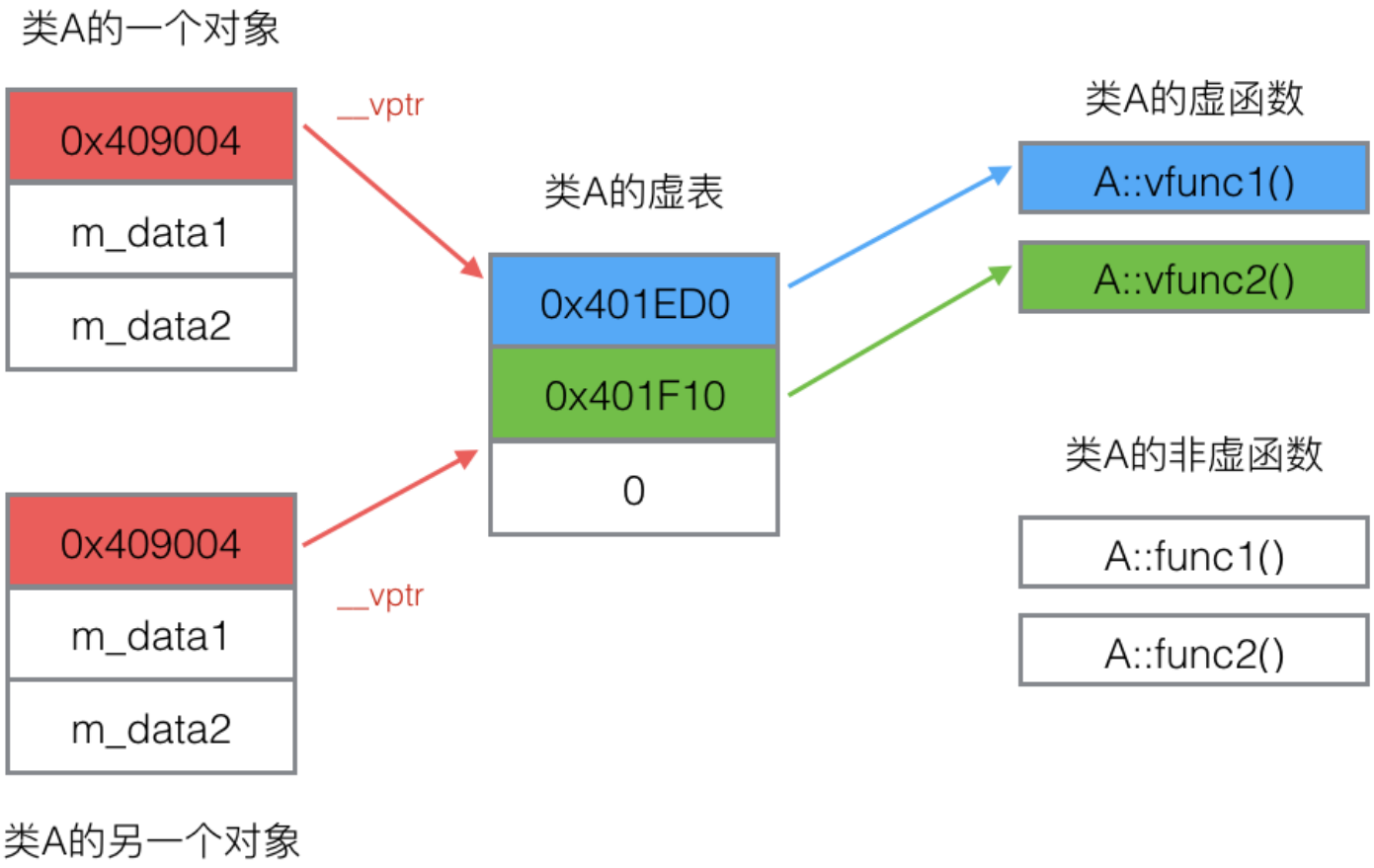


虚表是一个指针数组，其元素是虚函数的指针，每个元素对应一个虚函数的函数指针。需要指出的是，普通的函数即非虚函数，其调用并不需要经过虚表，所以虚表的元素并不包括普通函数的函数指针。

虚表内的条目，即虚函数指针的赋值发生在编译器的编译阶段，也就是说在代码的编译阶段，虚表就可以构造出来了。

虚表指针

虚表是属于类的，而不是属于某个具体的对象，一个类只需要一个虚表即可。同一个类的所有对象都使用同一个虚表。
为了指定对象的虚表，对象内部包含一个虚表的指针，来指向自己所使用的虚表。为了让每个包含虚表的类的对象都拥有一个虚表指针，编译器在类中添加了一个指针，*__vptr，用来指向虚表。这样，当类的对象在创建时便拥有了这个指针，且这个指针的值会自动被设置为指向类的虚表。

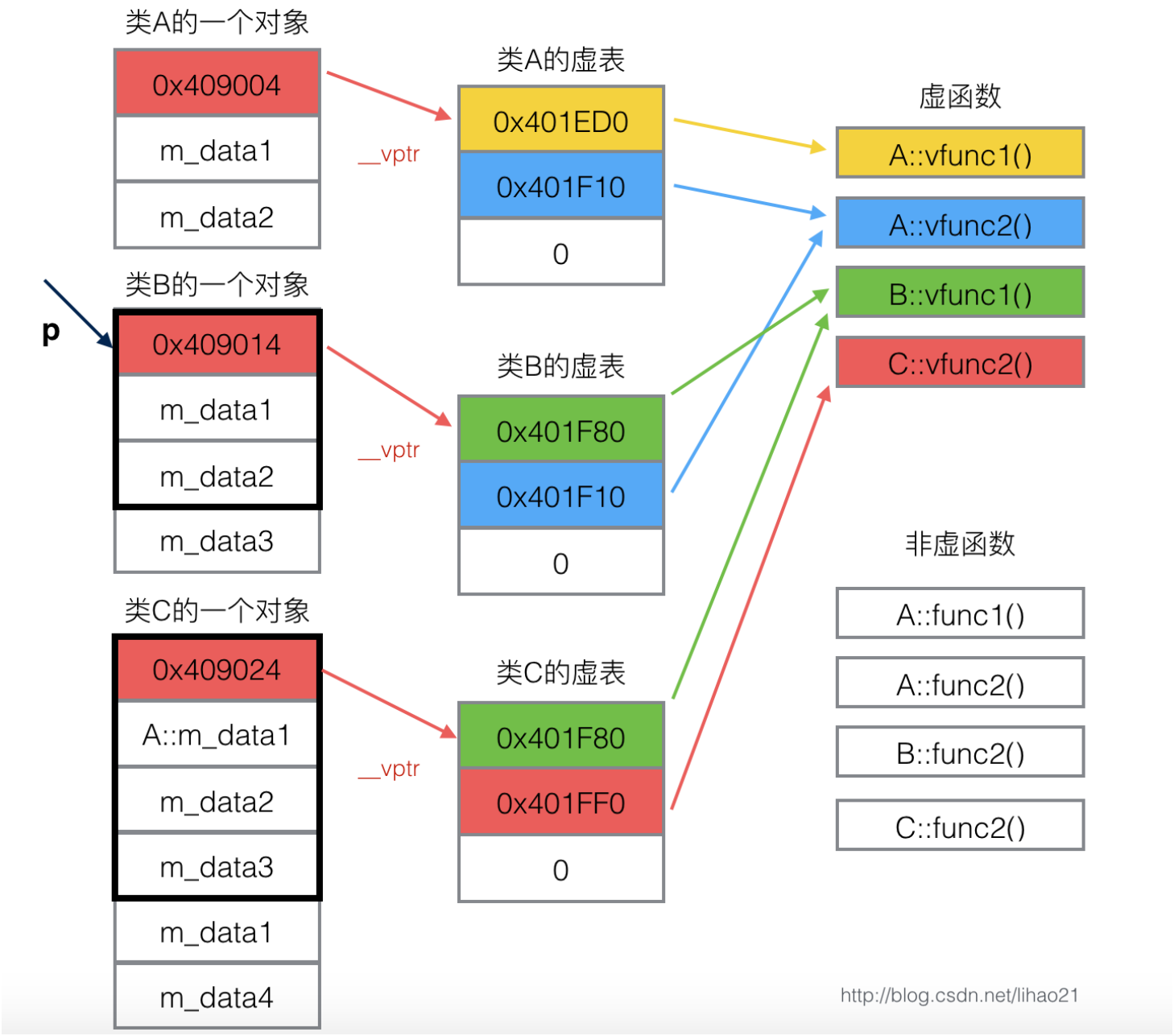


动态绑定

如何利用虚表和虚表指针来实现动态绑定：

```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};

class B : public A {
public:
    virtual void vfunc1();
    void func1();
private:
    int m_data3;
};

class C: public B {
public:
    virtual void vfunc2();
    void func2();
private:
    int m_data1, m_data4;
};
```

类A是基类，类B继承类A，类C又继承类B。类A，类B，类C，其对象模型如下图所示。



由于这三个类都有虚函数，故编译器为每个类都创建了一个虚表，即类A的虚表（A vtbl），类B的虚表（B vtbl），类C的虚表（C vtbl）。类A，类B，类C的对象都拥有一个虚表指针，*__vptr，用来指向自己所属类的虚表。

类A包括两个虚函数，故A vtbl包含两个指针，分别指向A::vfunc1()和A::vfunc2()。

类B继承于类A，故类B可以调用类A的函数，但由于类B重写了B::vfunc1()函数，故B vtbl的两个指针分别指向B::vfunc1()和A::vfunc2()。

类C继承于类B，故类C可以调用类B的函数，但由于类C重写了C::vfunc2()函数，故C vtbl的两个指针分别指向B::vfunc1()（指向继承的最近的一个类的函数）和C::vfunc2()。

假设我们定义一个类B的对象。由于bObject是类B的一个对象，故bObject包含一个虚表指针，指向类B的虚表。

```
int main()
{
    B bObject;
}
```

我们声明一个类A的指针p来指向对象bObject。虽然p是基类的指针只能指向基类的部分，但是虚表指针亦属于基类部分，所以p可以访问到对象bObject的虚表指针。bObject的虚表指针指向类B的虚表，所以p可以访问到B vtbl。

```
int main()
{
    B bObject;
    A *p = & bObject;
}
```

当我们使用p来调用vfunc1()函数时，会发生什么现象？

```
int main()
{
    B bObject;
    A *p = & bObject;
    p->vfunc1();
}
```

程序在执行p->vfunc1()时，会发现p是个指针，且调用的函数是虚函数，接下来便会进行以下的步骤。
首先，根据虚表指针p->__vptr来访问对象bObject对应的虚表。虽然指针p是基类A*类型，但是*__vptr也是基类的一部分，所以可以通过p->__vptr可以访问到对象对应的虚表。
然后，在虚表中查找所调用的函数对应的条目。由于虚表在编译阶段就可以构造出来了，所以可以根据所调用的函数定位到虚表中的对应条目。对于 p->vfunc1()的调用，B vtbl的第一项即是vfunc1对应的条目。
最后，根据虚表中找到的函数指针，调用函数。从图中可以看到，B vtbl的第一项指向B::vfunc1()，所以p->vfunc1()实质会调用B::vfunc1()函数。
如果p指向类A的对象，情况又是怎么样？

```
int main()
{
    A aObject;
    A *p = &aObject;
    p->vfunc1();
}
```

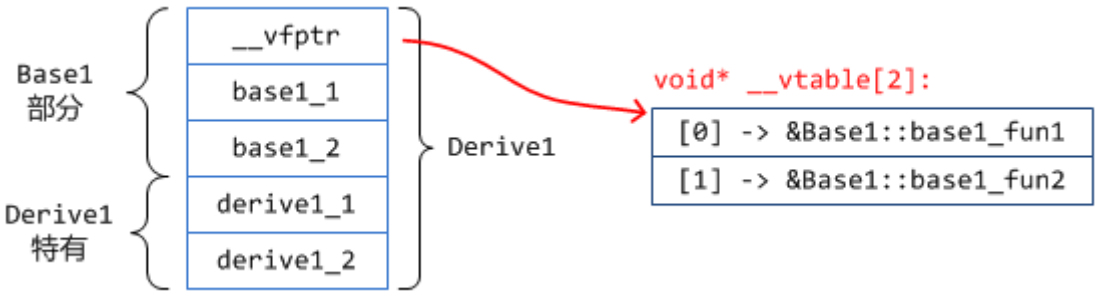
当aObject在创建时，它的虚表指针__vptr已设置为指向A vtbl，这样p->__vptr就指向A vtbl。vfunc1在A vtbl对应在条目指向了A::vfunc1()函数，所以 p->vfunc1()实质会调用A::vfunc1()函数。

单继承情况

单继承情况且本身不存在虚函数

```
class Base1
{
public:
    int base1_1;
    int base1_2;
    virtual void base1_fun1() {}
    virtual void base1_fun2() {}
};

class Derive1 : public Base1
{
public:
    int derive1_1;
    int derive1_2;
};
```

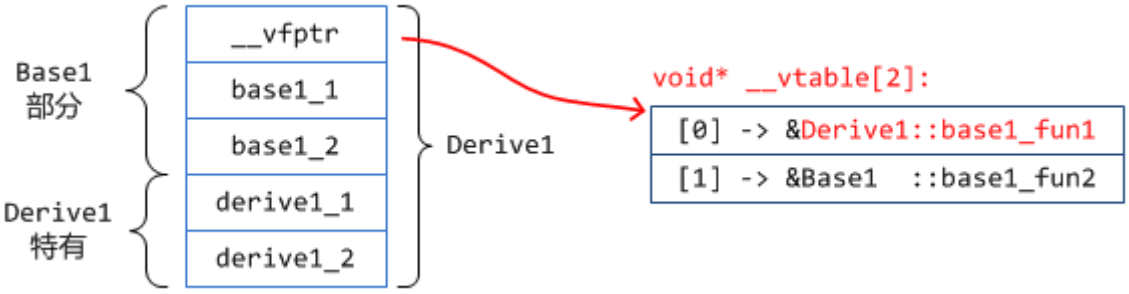
现在类的布局情况应该是下面这样:



单继承覆盖基类的虚函数

```
class Base1
{
public:
    int base1_1;
    int base1_2;
    virtual void base1_fun1() {}
    virtual void base1_fun2() {}
};

class Derive1 : public Base1
{
public:
    int derive1_1;
    int derive1_2;
    // 覆盖基类函数
    virtual void base1_fun1() {}
};
```

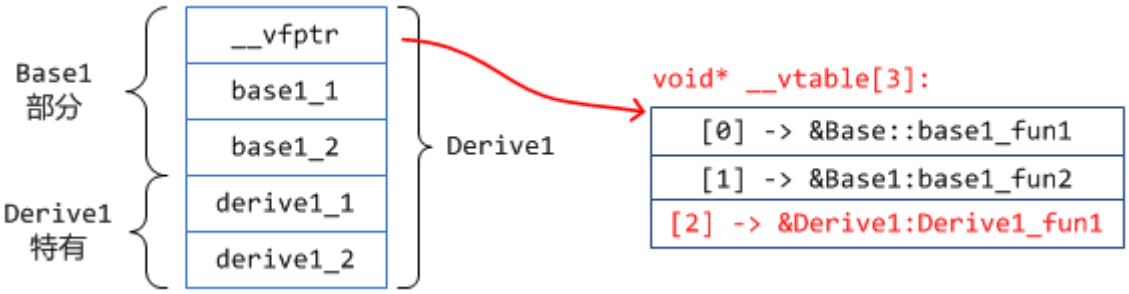
Derive1类 重写了Base1类的base1_fun1()函数, 也就是常说的虚函数覆盖。无论是通过Derive1的指针还是Base1的指针来调用此方法, 调用的都将是被继承类重写后的那个方法(函数), 这时就产生了多态。那么新的布局图:



单继承同时新定义了基类没有的虚函数

```
class Base1
{
public:
    int base1_1;
    int base1_2;
    virtual void base1_fun1() {}
    virtual void base1_fun2() {}
};

class Derive1 : public Base1
{
public:
    int derive1_1;
    int derive1_2;
    virtual void derive1_fun1() {}
};
```

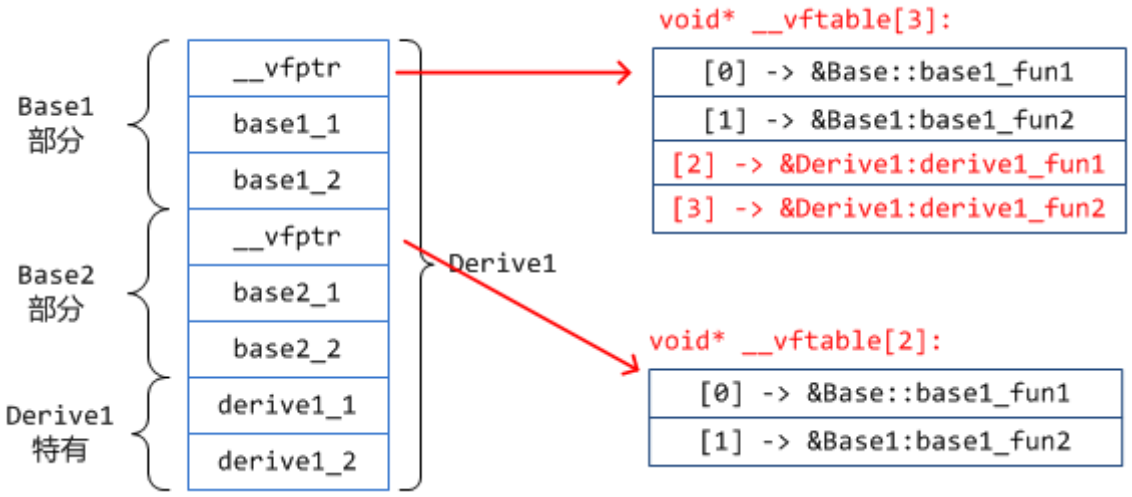
继承类Derive1的虚函数表被加在基类的后面。
由于Base1只知道自己的两个虚函数索引[0][1], 所以就算在后面加上了[2], Base1根本不知情, 不会对她造成任何影响。



多继承且存在虚函数覆盖又存在自身定义的虚函数的类对象布局

```
class Base1
{
public:
    int base1_1;
    int base1_2;
    virtual void base1_fun1() {}
    virtual void base1_fun2() {}
};
class Base2
{
public:
    int base2_1;
    int base2_2;
    virtual void base2_fun1() {}
    virtual void base2_fun2() {}
};
// 多继承
class Derive1 : public Base1, public Base2
{
public:
    int derive1_1;
    int derive1_2;
    // 基类虚函数覆盖
    virtual void base1_fun1() {}
    virtual void base2_fun2() {}
    // 自身定义的虚函数
    virtual void derive1_fun1() {}
    virtual void derive1_fun2() {}
};
```

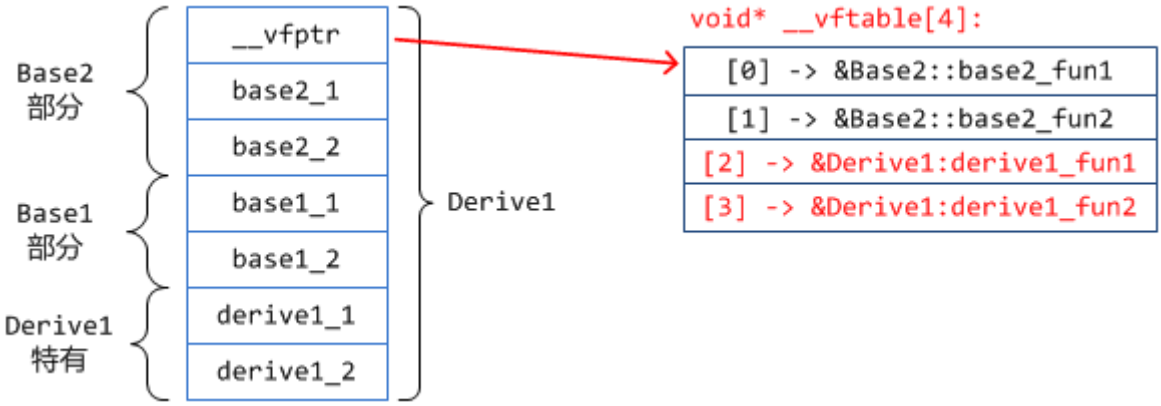
Derive1的虚函数表依然是保存到第1个拥有虚函数表的那个基类的后面的.
看看现在的类对象布局图:



如果第**1**个直接基类没有虚函数

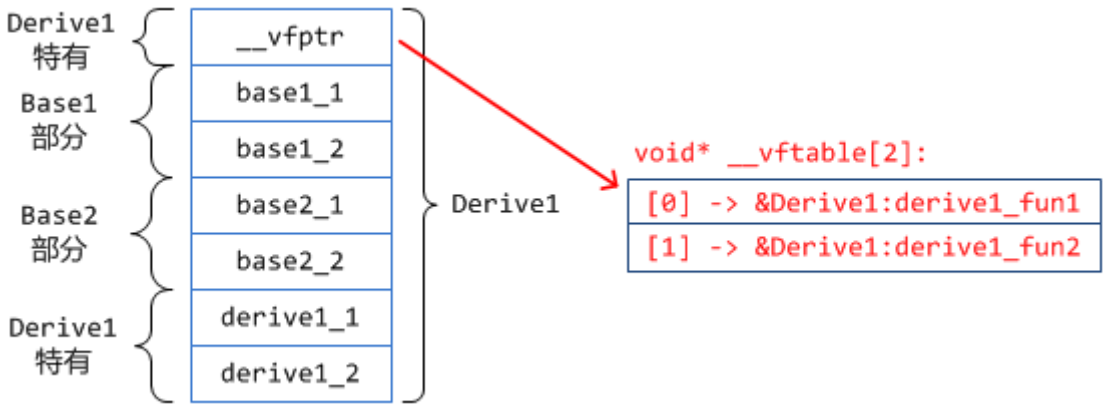
```
class Base1
{
public:
    int base1_1;
    int base1_2;
};
class Base2
{
public:
    int base2_1;
    int base2_2;
    virtual void base2_fun1() {}
    virtual void base2_fun2() {}
};
// 多继承
class Derive1 : public Base1, public Base2
{
public:
    int derive1_1;
    int derive1_2;
    // 自身定义的虚函数
    virtual void derive1_fun1() {}
    virtual void derive1_fun2() {}
};
```

谁有虚函数表, 谁就放在前面!



What if 两个基类都没有虚函数表

```
class Base1
{
public:
    int base1_1;
    int base1_2;
};
class Base2
{
public:
    int base2_1;
    int base2_2;
};
// 多继承
class Derive1 : public Base1, public Base2
{
public:
    int derive1_1;
    int derive1_2;
    // 自身定义的虚函数
    virtual void derive1_fun1() {}
    virtual void derive1_fun2() {}
};
```



面试题

一个空类编译器会自动生成哪些函数？ 哪些需要禁止？

当空类Empty_one定义一个对象时Empty_one pt;sizeof(pt)仍是为1，但编译器会生成6个成员函数：一个缺省的构造函数、一个拷贝构造函数、一个析构函数、一个赋值运算符、两个取址运算符。

```
class Empty
{
  public:
    Empty();                                //缺省构造函数
    Empty(const Empty &rhs);                //拷贝构造函数
    ~Empty();                               //析构函数
    Empty& operator=(const Empty &rhs); //赋值运算符
    Empty* operator&();                     //取址运算符
    const Empty* operator&() const;     //取址运算符(const版本)
};
```

对于某些类而言，对象的拷贝或赋值时不合法的，例如定义了一个学生类，但对于学生对象而言，只能有一个，世界上不存在两个一样的学生对象，我们应该明确阻止学生对象之间的拷贝或赋值，也就是说学生类是不支持拷贝或赋值的。
阻止拷贝构造函数及拷贝赋值运算符的生成，下面主要介绍三种：
1、在C++11标准下，将这些函数声明为删除的函数，在函数参数的后面加上=delete来指示出我们定义的删除的函数
2、将这些函数声明为private,并且不提供函数定义
3、将待定义的类成为一个不支持copy的类的子类

如何限制一个类对象只在栈（堆）上分配空间？

在C++中，类的对象建立分为两种，一种是静态建立，如A a；另一种是动态建立，如A* ptr=new A；这两种方式是有区别的。
静态建立类对象：
是由编译器为对象在栈空间中分配内存，是通过直接移动栈顶指针，挪出适当的空间，然后在这片内存空间上调用构造函数形成一个栈对象。使用这种方法，直接调用类的构造函数。
动态建立类对象：
是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间。这种方法，间接调用类的构造函数。
只能在堆上分配类对象
当对象建立在栈上面时，是由编译器分配内存空间的，调用构造函数来构造栈对象。当对象使用完后，编译器会调用析构函数来释放栈对象所占的空间。编译器管理了对象的整个生命周期。如果编译器无法调用类的析构函数，情况会是怎样的呢？比如，类的析构函数是私有的，编译器无法调用析构函数来释放内存。所以，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性，其实不光是析构函数，只要是非静态的函数，编译器都会进行检查。如果类的析构函数是私有的，则编译器不会在栈空间上为类对象分配内存。
因此，将**析构函数设为私有**，类对象就无法建立在栈上了。

```
class A
{
public:
    A(){}
    void destory(){delete this;}
private:
    ~A(){}
};
```

试着使用A a;来建立对象，编译报错，提示析构函数无法访问。这样就只能使用new操作符来建立对象，构造函数是公有的，可以直接调用。类中必须提供一个destory函数，来进行内存空间的释放。类对象使用完成后，必须调用destory函数。

上述方法的缺点：

一、无法解决继承问题。

如果A作为其它类的基类，则析构函数通常要设为virtual，然后在子类重写，以实现多态。

因此析构函数不能设为private。

还好C++提供了第三种访问控制，protected。

将析构函数设为protected可以有效解决这个问题，类外无法访问protected成员，子类则可以访问。

二、类的使用很不方便，

使用new建立对象，却使用destory函数释放对象，而不是使用delete。

(使用delete会报错，因为delete对象的指针，会调用对象的析构函数，而析构函数类外不可访问。这种使用方式比较怪异。)

为了统一，可以将构造函数设为protected，然后提供一个public的static函数来完成构造，这样不使用new，而是使用一个函数来构造，使用一个函数来析构。

```
class A
{
protected:
    A(){}
    ~A(){}
public:
    static A* create()
    {
        return new A();
    }
    void destory()
    {
        delete this;
    }
};
```

这样，调用create()函数在堆上创建类A对象，调用destory()函数释放内存。

只能在栈上分配类对象

只有使用new运算符，对象才会建立在堆上，因此，只要禁用new运算符就可以实现类对象只能建立在栈上。

虽然你不能影响new operator的能力（因为那是C++语言内建的），但是你可以利用一个事实：new operator 总是先调用 operator new，而后者我们是可以自行声明重写的。

因此，将operator new()设为私有即可禁止对象被new在堆上。

```
class A
{
private:
    void* operator new(size_t t){}      // 注意函数的第一个参数和返回值都是固定的
    void operator delete(void* ptr){} // 重载了new就需要重载delete
public:
    A(){}
    ~A(){}
};
```

类静态成员函数的特点、静态成员函数可以是虚函数吗、静态成员函数可以是const函数吗？

它为类的全部服务，而不是为某一个类的具体对象服务。静态成员函数与静态数据成员一样，都是在类的内部实现，属于类定义的一部分。普通的成员函数一般都隐藏了一个this指针，this指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，this指针是缺省的、但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有this指针，从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能调用其余的静态成员函数。

特点：

1.出现在类体外的函数不能指定关键字static；

2.静态成员之间可以互相访问，包括静态成员函数访问静态数据成员和访问静态成员函数；

3.非静态成员函数可以任意地访问静态成员函数和静态数据成员；

4.静态成员函数不能访问非静态成员函数和非静态数据成员

5.由于没有this指针的额外开销，因此静态成员函数与类的全局函数相比，速度上会有少许的增长

6.调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指调用静态成员函数。

不能为虚函数。

1. static成员不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义的。
2. 静态与非静态成员函数之间有一个主要的区别。那就是静态成员函数没有this指针。
   虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，因为它是类的一个成员，并且vptr指向保存虚函数地址的vtable.

对于静态成员函数，它没有this指针，所以无法访问vptr. 这就是为何static函数不能为virtual.
虚函数的调用关系：this -> vptr -> vtable ->virtual function

不能为**const**。
静态成员函数是属于类的，而不是某个具体对象，在没有具体对象的时候静态成员就已经存在，静态成员函数不会访问到非静态成员，也不存在this指针。而成员函数的const就是修饰this指针的，既然静态成员函数不会被传递this指针，那const自然就没有必要了

不能为**volatile**
volatile关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。声明时语法：int volatile vInt; 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。
与const同理

this指针

this指针指向类的某个实例（对象），叫它当前对象。在成员函数执行的过程中，正是通过“This指针”才能找到对象所在的地址，因而也就能找到对象的所有非静态成员变量的地址。

友元函数

为什么要有？

友元函数是一个不属于类成员的函数，但它可以访问该类的私有成员。换句话说，友元函数被视为好像是该类的一个成员。友元函数可以是常规的独立函数，也可以是其他类的成员。实际上，整个类都可以声明为另一个类的友元。

为了使一个函数或类成为另一个类的友元，必须由授予它访问权限的类来声明。类保留了它们的朋友的"名单"，只有名字出现在列表中的外部函数或类才被授予访问权限。通过将关键字 friend 放置在函数的原型之前，即可将函数声明为友元。

使用友元函数的优缺点

优点：能够提高效率，表达简单、清晰。
缺点：友元函数破坏了封装机制，尽量不使用成员函数，除非不得已的情况下才使用友元函数。

友元函数的使用

可以直接调用友元函数，不需要通过对象或指针;此外，友元函数没有this指针，则参数要有三种情况：
1、要访问非static成员时，需要对象做参数；
2、要访问static成员或全局变量时，则不需要对象做参数；
3、如果做参数的对象是全局对象，则不需要对象做参数.

```
class Box{
        double width;  // 默认是private
        public:
            double length;
            friend void printWidth(Box box);  // 友元函数声明
            friend class BigBox;  // 友元类的声明
            void setWidth(double wid);
    };

    // 成员函数的定义
    void Box::setWidth(double wid){
        width = wid;
    }
    // 友元函数的定义
    // 请注意：printWidth() 不是任何类的成员函数!!!
    void printWidth(Box box){
        /* 因为printWidth()是Box的友元函数，它可以直接访问该类的任何成员 */
        cout << "Width of Box: " << box.width << endl;
    }

    // 友元类的使用
    class BigBox{
        public:
            void Print(int width, Box &box){
                // BigBox是Box类的友元类，它可以直接访问Box类的任何成员
                box.setWidth(width);
                cout << "Width of Box: " << box.width << endl;
            }
    };

    int main(){
        Box box;
        BigBox big;
        // 使用成员函数设置宽度
        box.setWidth(10.0);
        // 使用友元函数输出宽度
        printWidth(box); // 调用友元函数!
        cout << "-------------------------------------\n";
        // 使用友元类中的方法设置宽度
        big.Print(20, box);
        return 0;
    }
```

C++中哪些函数不可以是虚函数?

1、**普通函数（非成员函数）**：我在前面多态这篇博客里讲到，定义虚函数的主要目的是为了重写达到多态，所以普通函数声明为虚函数没有意义，因此编译器在编译时就绑定了它。

2、**静态成员函数**：静态成员函数对于每个类都只有一份代码，所有对象都可以共享这份代码，他不归某一个对象所有，所以它也没有动态绑定的必要。

3、**内联成员函数**：内联函数本就是为了减少函数调用的代价，所以在代码中直接展开。但虚函数一定要创建虚函数表，这两者不可能统一。另外，内联函数在编译时被展开，而虚函数在运行时才动态绑定。

4、**构造函数**：
1）因为创建一个对象时需要确定对象的类型，而虚函数是在运行时确定其类型的。而在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型，是类本身还是类的派生类等等

2）虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数即构造函数了

5、**友元函数**：当我们把一个函数声明为一个类的友元函数时，它只是一个可以访问类内成员的普通函数，并不是这个类的成员函数，自然也不能在自己的类内将它声明为虚函数。

C 指针指向的是物理地址吗？

c/c++的指针是指向逻辑地址。
以windows平台为例，任何一个C++程序肯定是运行在某一个进程中，windows的32位系统对每一个用户进程都管理着一个寻址范围为4GB的地址空间, 各个进程的地址空间是相互独立的，很显然这是一个逻辑的地址空间，C++指针指向进程内的一个逻辑内存地址，然后由操作系统管理着从逻辑地十到物理地址的映射。

模板的编译过程，模板是什么时候实例化的？模板特化·

编译过程

通常我们将函数或类的声明放在头(.h)文件中，定义放在(.cpp)文件中，在其他文件中使用该函数或类时引用头文件即可，编译器是怎么工作的呢？编译器首先编译所有cpp文件，如果在程序中用到某个函数或类，只是判断这个函数或类是否已经声明，并不会立即找到这个函数或类定义的地址，只有在链接的过程中才会去寻找具体的地址，所以我们如果只是对某个函数或类声明了，而不定义它的具体内容，如果我不再其他地方使用它，这事没有任何问题的。
而编译器在编译模板所在的文件时，模板的内容是不会立即生成二进制代码的，直到有地方使用到该模板时，才会对该模板生成二进制代码（即模板实例化）。但是，如果我们将模板的声明部分和实现部分分别放在.h和.cpp两个文件中时，问题就出现了：由于模板的cpp文件中使用的不是具体类型，所以编译器不能为其生成二进制代码，在其他文件使用模板时只是引用了头文件，编译器在编译时可以识别该模板，编译可以通过；但是在链接时就不行了，二进制代码根本就没有生成，链接器当然找不到模板的二进制代码的地址了，就会出现找不到函数地址类似的错误信息了。
所以，在通常情况下，我们在定义模板时将声明和定义都放在了头文件中，STL就是这样。当然了，也

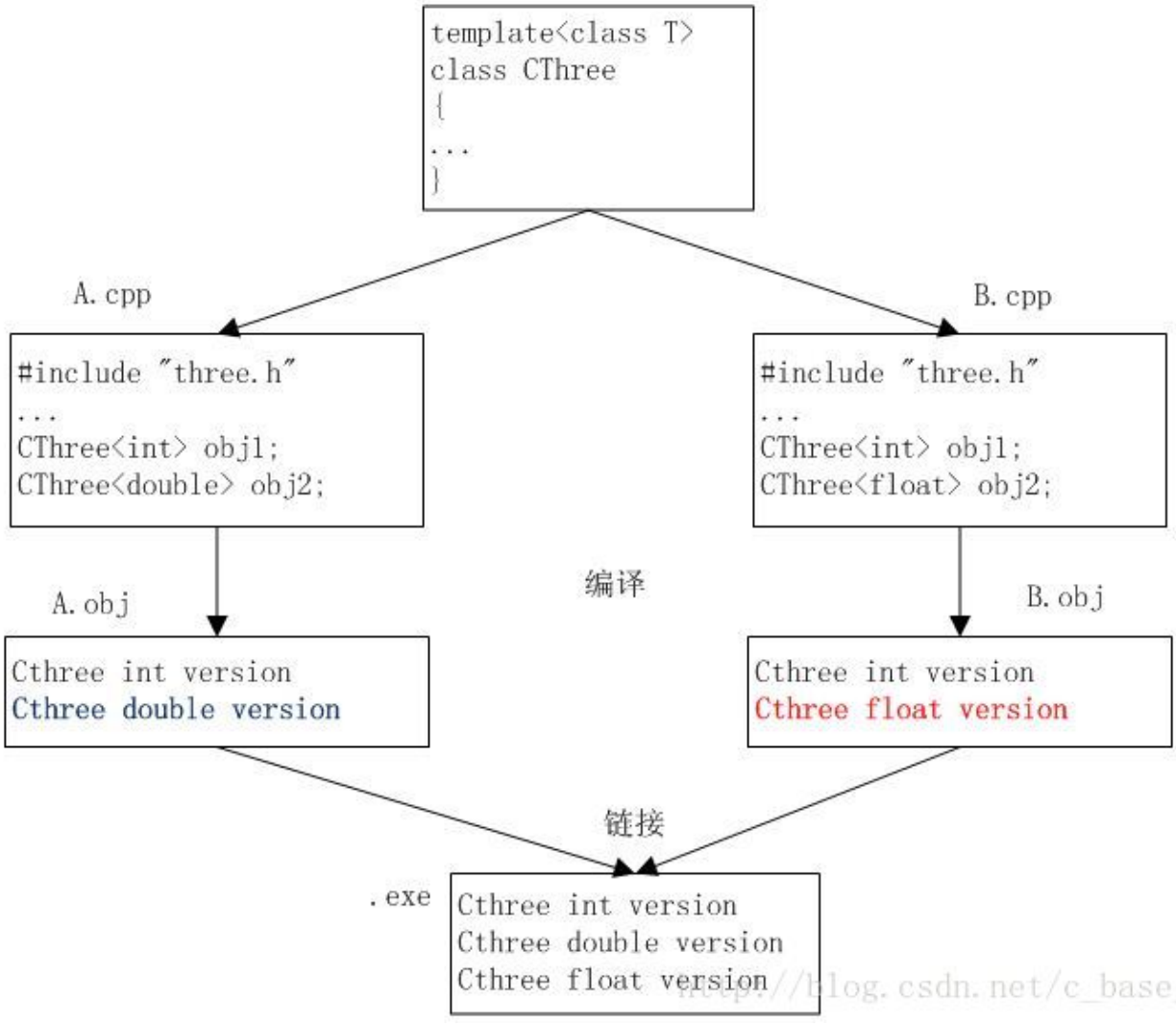
可以像C++ Primer中所述的，使用export关键字。（你可以在.h文件中，声明模板类和模板函数；在.cpp文件中，使用关键字export来定义具体的模板类对象和模板函数；然后在其他用户代码文件中，包含声明头文件后，就可以使用该这些对象和函数了）

编译和链接：

当编译器遇到一个template时，不能够立马为他产生机器代码，它必须等到template被指定某种类型。也就是说，函数模板和类模板的完整定义将出现在template被使用的每一个角落。

对于不同的编译器，其对模板的编译和链接技术也会有所不同，其中一个常用的技术称之为Smart，其基本原理如下：

1. 模板编译时，以每个cpp文件为编译单位，实例化该文件中的函数模板和类模板
2. 链接器在链接每个目标文件时，会检测是否存在相同的实例；有存在相同的实例版本，则删除一个重复的实例，保证模板实例化没有重复存在。
   比如我们有一个程序，包含A.cpp和B.cpp，它们都调用了CThree模板类，在A文件中定义了int和double型的模板类，在B文件中定义了int和float型的模板类；在编译器编译时.cpp文件为编译基础，生成A.obj和B.obj目标文件，即使A.obj和B.obj存在重复的实例版本，但是在链接时，链接器会把所有冗余的模板实例代码删除，保证exe中的实例都是唯一的。编译原理和链接原理，如下所示：



实例化

在我们使用类模板时，只有当代码中使用了类模板的一个实例的名字，而且上下文环境要求必须存在类的定义时，这个类模板才被实例化。

1. 声明一个类模板的指针和引用，不会引起类模板的实例化，因为没有必要知道该类的定义。
2. 定义一个类类型的对象时需要该类的定义，因此类模板会被实例化。
3. 在使用sizeof()时，它是计算对象的大小，编译器必须根据类型将其实例化出来，所以类模板被实例化.
4. new表达式要求类模板被实例化。
5. 引用类模板的成员会导致类模板被编译器实例化。
6. 需要注意的是，类模板的成员函数本身也是一个模板。标准C++要求这样的成员函数只有在被调用或者取地址的时候，才被实例化。用来实例化成员函数的类型，就是其成员函数要调用的那个类对象的类型

模板的特化

一是特化为绝对类型（全特化）； 二是特化为引用，指针类型(半特化、偏特化)

模板函数只能全特化
模板类都可以

全特化，就是模板中参数全被指定为确定的类型。
全特化就是定义了一个全新的类型，全特化的类中的函数可以与模板类不一样。
偏特化：就是模板中的模板参数没有被全部确定，需要编译器在编译时进行确定。
在类型中加上const，&，（*const int int& int* 等等），并没有产生新的类型，只是类型被修饰了。模板在编译时，可以得到这些修饰信息。

全特化的标志就是产生出完全确定的东西，而不是还需要在编译期间去搜寻合适的特化实现，全特化的东西无论是类还是函数都有该特点。
一个特化的模板类的标志：在定义类实现时，加上了<>
比如 class A <int T>; 但是在定义一个模板类的时候，class A后面是没有<>的。
全特化的标志：template<> 然后是完全和模板类型没有一点关系的类实现或者函数定义。
偏特化的标志：
template<typename T,...>还剩点东西，不像全特化那么彻底。

运算符重载

重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

```
Box operator+(const Box&);
```

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

```
Box operator+(const Box&, const Box&);
```

例子：

```
#include <iostream>
using namespace std;
class complex{
public:
    complex();
    complex(double real, double imag);
public:
    //声明运算符重载
    complex operator+(const complex &A) const;
    void display() const;
private:
    double m_real;  //实部
    double m_imag;  //虚部
};
complex::complex(): m_real(0.0), m_imag(0.0){ }
complex::complex(double real, double imag): m_real(real), m_imag(imag){ }
//实现运算符重载
complex complex::operator+(const complex &A) const{
    complex B;
    B.m_real = this->m_real + A.m_real;
    B.m_imag = this->m_imag + A.m_imag;
    return B;
}
void complex::display() const{
    cout<<m_real<<" + "<<m_imag<<"i"<<endl;
}
int main(){
    complex c1(4.3, 5.8);
    complex c2(2.4, 3.7);
    complex c3;
    c3 = c1 + c2;
    c3.display();
    return 0;
}
```

我们在 complex 类中重载了运算符+，该重载只对 complex 对象有效。当执行c3 = c1 + c2;语句时，编译器检测到+号左边（+号具有左结合性，所以先检测左边）是一个 complex 对象，就会调用成员函数

operator+()，也就是转换为下面的形式：

```
c3 = c1.operator+(c2);
```

c1 是要调用函数的对象，c2 是函数的实参。
运算符重载函数不仅可以作为类的成员函数，还可以作为全局函数。更改上面的代码，在全局范围内重载+，实现复数的加法运算：

```
#include <iostream>
using namespace std;
class complex{
public:
    complex();
    complex(double real, double imag);
public:
    void display() const;
    //声明为友元函数
    friend complex operator+(const complex &A, const complex &B);
private:
    double m_real;
    double m_imag;
};
complex operator+(const complex &A, const complex &B);
complex::complex(): m_real(0.0), m_imag(0.0){ }
complex::complex(double real, double imag): m_real(real), m_imag(imag){ }
void complex::display() const{
    cout<<m_real<<" + "<<m_imag<<"i"<<endl;
}
//在全局范围内重载+
complex operator+(const complex &A, const complex &B){
    complex C;
    C.m_real = A.m_real + B.m_real;
    C.m_imag = A.m_imag + B.m_imag;
    return C;
}
int main(){
    complex c1(4.3, 5.8);
    complex c2(2.4, 3.7);
    complex c3;
    c3 = c1 + c2;
    c3.display();
    return 0;
}
```

运算符重载函数不是 complex 类的成员函数，但是却用到了 complex 类的 private 成员变量，所以必须在 complex 类中将该函数声明为友元函数。

当执行c3 = c1 + c2;语句时，编译器检测到+号两边都是 complex 对象，就会转换为类似下面的函数调用：

```
c3 = operator+(c1, c2);
```

运算符重载规则

1. 不是所有的运算符都可以重载。长度运算符sizeof、条件运算符: ?、成员选择符.和域解析运算符::不能被重载。
2. 重载不能改变运算符的优先级和结合性。
3. 重载不会改变运算符的用法。
4. 运算符重载函数不能有默认的参数，否则就改变了运算符操作数的个数。
5. 运算符函数既可以作为类的成员函数，也可以作为全局函数。
6. 箭头运算符->、下标运算符[ ]、函数调用运算符( )、赋值运算符=只能以成员函数的形式重载。

到底以成员函数还是全局函数（友元函数）的形式重载运算符

（1）一般而言，对于双目运算符，最好将其重载为友元函数；而对于单目运算符，则最好重载为成员函数。
但是也存在例外情况。有些双目运算符是不能重载为友元函数的，比如赋值运算符=、函数调用运算符()、下标运算符[]、指针运算符->等，因为这些运算符在语义上与this都有太多的关联。比如=表示“将自身赋值为…”，[]表示“自己的第几个元素”，如果将其重载为友元函数，则会出现语义上的不一致。

2）还有一个需要特别说明的就是输出运算符<<。因为<<的第一个操作数一定是ostream类型，所以<<只能重载为友元函数，如下：

```
friend ostream& operator <<(ostream& os, const Complex& c);
ostream& operator <<(ostream& os, const Complex& c)
{
    os << c.m_Real << “+” << c.m_Imag << “i” << endl;
    return os;
}
```

（3）所以，对于 =、[]、()、->以及所有的类型转换运算符只能作为非静态成员函数重载。如果允许第一操作数不是同类对象，而是其他数据类型，则只能作为非成员函数重载（如输入输出流运算符>>和<<就是这样的情况）。

重载[]（下标运算符）

下标运算符[ ]必须以成员函数的形式进行重载。该重载函数在类中的声明格式如下：

```
返回值类型 & operator[ ] (参数);
```

或者：

```
const 返回值类型 & operator[ ] (参数) const;
```

使用第一种声明方式，[ ]不仅可以访问元素，还可以修改元素。使用第二种声明方式，[ ]只能访问而不能修改元素。在实际开发中，我们应该同时提供以上两种形式，这样做是为了适应 const 对象，因为通过 const 对象只能调用 const 成员函数，如果不提供第二种形式，那么将无法访问 const 对象的任何元素。

```
#include <iostream>
using namespace std;
class Array{
public:
    Array(int length = 0);
    ~Array();
public:
    int & operator[](int i);
    const int & operator[](int i) const;
public:
    int length() const { return m_length; }
    void display() const;
private:
    int m_length;  //数组长度
    int *m_p;  //指向数组内存的指针
};
Array::Array(int length): m_length(length){
    if(length == 0){
        m_p = NULL;
    }else{
        m_p = new int[length];
    }
}
Array::~Array(){
    delete[] m_p;
}
int& Array::operator[](int i){
    return m_p[i];
}
const int & Array::operator[](int i) const{
    return m_p[i];
}
void Array::display() const{
    for(int i = 0; i < m_length; i++){
        if(i == m_length - 1){
            cout<<m_p[i]<<endl;
        }else{
            cout<<m_p[i]<<", ";
        }
    }
}
int main(){
    int n;
    cin>>n;
    Array A(n);
    for(int i = 0, len = A.length(); i < len; i++){
        A[i] = i * 5;
    }
    A.display();

    const Array B(n);
```

```
    cout<<B[n-1]<<endl;  //访问最后一个元素

    return 0;
}
```

重载[ ]运算符以后，表达式arr[i]会被转换为：

```
arr.operator[ ](i);
```

需要说明的是，B 是 const 对象，如果 Array 类没有提供 const 版本的operator[ ]，那么倒数第二行代码将报错。虽然这行代码只是读取对象的数据，并没有试图修改对象，但是它调用了非 const 版本的 operator[ ]，编译器不管实际上有没有修改对象，只要是调用了非 const 的成员函数，编译器就认为会修改对象（至少有这种风险）。

赋值运算符重载

```
#include <iostream>
#include <string>
using namespace std;

class MyStr {
public:
    MyStr() {}
    MyStr(int _id, char *_name)
    {
        cout << "constructor" << endl;
        id = _id;
        name = new char[strlen(_name) + 1];
        strcpy_s(name,strlen(_name) + 1,_name);
    }
    MyStr(const MyStr &str)
    {
        cout << "copy constructor" << endl;
        id = str.id;
        if (name != NULL)
            delete name;
        name = new char[strlen(str.name) + 1];
        strcpy_s(name,strlen(str.name) + 1,str.name);
    }

    MyStr& operator=(const MyStr& str)
    {
        cout << "operator=" << endl;
        if (this != &str)
        {
            if (name != NULL)
                delete name;
            this->id = str.id;
            name = new char[strlen(str.name) + 1];
            strcpy_s(name,strlen(str.name) + 1,str.name);

            return *this;
        }
    }
    ~MyStr()
    {
        cout << "deconstructor" << endl;
        delete name;
    }
private:
    char *name;
    int id;
};

void main()
{
    MyStr str1(1,"Jack");
```

```
    MyStr str2;
    str2 = str1;
    MyStr str3 = str2;
    return;

}
```

如果将上述例子显示提供的拷贝函数注释掉，然后同样执行MyStr str3 = str2;语句，此时调用默认的拷贝构造函数，它们指向内存中的同一区域。
这样会有两个致命错误：
1）str2修改name时，str3的name也会被修改；
2）当执行str2和str3的析构函数时，会导致同一内存区域释放两次，程序崩溃。
所以，必须通过显示提供拷贝构造函数以避免这样的问题，如上述例子，先判断被拷贝者的name是否为空，若否，delete name，然后为name重新申请空间，再将拷贝者name中的数据拷贝到被拷贝者的name中，这样，str2.name和str3.name各自独立，避免了上面两个错误。赋值运算符重载函数也是同样的道理。

赋值运算符重载函数只能是类的非静态的成员函数
1、因为静态成员函数只能操作类的静态成员，无法操作类的非静态成员，可以参考静态成员变量和静态成员函数在C++类中的作用来进行理解；
2、避免二义性
当程序没有显示提供一个以本类或者本类的引用为参数的赋值运算符重载函数时，编译器会自动提供一个。现在假设C++允许友元函数定义的赋值运算符重载函数，而且以引用为参数，与此同时，编译器也提供一个默认的赋值运算符重载函数（由于友元函数不属于这个类，所以此时编译器会自动提供一个）。但是当再执行类似str2 = str1;这样的代码时，编译器就困惑了。
为了避免这样的二义性，C++强制规定，赋值运算符重载函数只能定义为类的成员函数，这样编译器就能判断是否需要提供默认版本了。

取地址及const取地址操作符重载

取地址是什么意思呢？就是返回当前对象的地址，对于成员函数来讲，this指针就是它的地址，需要返回指针。
"&" 运算符是一个单目运算符，其只有一个参数，而这个参数就是一个对象，所以说这个对象是不用传的，定义为成员函数时函数参数就应该少一个，第一个函数参数就被this指针所代替。所以，在此不需要进行传参。

const成员函数及const对象去调用，普通的成员函数普通的对象来进行调用，若没有普通成员函数，那么普通对象也能够调用const成员函数。

```
class Date {
public:
	Date(int year, int month, int day) {
		_year = year;
		_month = month;
		_day = day;
	}
	Date(const Date& d) {
		_year = d._year;
	}
	Date* operator&() {
		cout << "Date* operator&()" << endl;
		return this;
	}

	const Date* operator&() const {
		cout << "const Date* operator&() const" << endl;
		return this;
	}

private:
	int _year;
	int _month;
	int _day;
};

int main() {
	Date d1(2019, 4, 1);
	const Date d2(2019, 3, 31);

	Date* pa1 = &d1;
	const Date* pd2 = &d2;
	system("pause");
	return 0;
}
```

如果不写这两个函数的时候，编译器会帮助默认生成，若无其它操作完全够用了，因为这两个函数只返回this指针，也没有其他的操作。除非，你想返回别的地址，可以做到"返回你想返回的地址"，比如，返回一个病毒的地址，返回一个很深的调用链等等，可以自己按照需求进行重载实现，否则不必实现也无影响。

Operator char()什么意思

operator用于类型转换函数：
类型转换函数的特征：

1. 型转换函数定义在源类中；
2. 须由 operator 修饰，函数名称是目标类型名或目标类名；

3. 函数没有参数，没有返回值，但是有return 语句，在return语句中返回目标类型数据或调用目标类的构造函数。

   类型转换函数主要有两类：

1） 对象向基本数据类型转换：

```
class D {
  public:
     D(double d) : d_(d) {}

     /* “(int)D”类型转换:将类型D转换成int */
     operator int() const {
         std::cout << "(int)d called!" << std::endl;
         return static_cast<int>(d_);
     }

   private:
     double d_;
 };

 int add(int a, int b) {
     return a + b;
 }

  int main() {
     D d1 = 1.1;
     D d2 = 2.2;
     std::cout << add(d1, d2) << std::endl;
     return 0;
 }
```

执行add(d1,d2)函数时“(int)D”类型转换函数将被自动调用，程序运行的输出为：

(int)d called!

(int)d called!

3

2）对象向不同类的对象的转换：

```
class A
{
  public:
     A(int num = 0) : dat(num) {}

     /* "(int)a"类型转换 */
     operator int() { return dat; }

  private:
     int dat;
};


class X
{
  public:
     X(int num = 0) : dat(num) {}

     /* "(int)a"类型转换 */
     operator int() { return dat; }

     /* "(A)a"类型转换 */
     operator A() {
         A temp = dat;
         return temp;
     }

  private:
     int dat;
};


int main()
{
     X stuff = 37;
     A more = 0;
     int hold;

     hold = stuff;    // convert X::stuff to int
     std::cout << hold << std::endl;

     more = stuff;    // convert X::stuff to A::more
     std::cout << more << std::endl;        // convert A::more to int

     return 0;
}
```

上面这个程序中X类通过“operator A()”类型转换来实现将X类型对象转换成A类型，这种方式需要先创建一个临时A对象再用它去赋值目标对象；更好的方式是为A类增加一个构造函数：

```
A(const X& rhs) : dat(rhs) {}
```

定位内存泄露

不用加减乘除求一个数的7倍

两个鸡蛋，100层楼，判断出鸡蛋会碎的临界层