- 面向对象
  - C和C++的主要区别
  - 面向对象与面向过程分别是什么？
  - 面向过程与面向对象的优缺点？
  - struct和class的区别：
  - 哪种使用的情况下适合结构体还是类？
  - 多态
  - 重载、覆盖（重写）、隐藏（重定义）
- 类
  - 定义
  - 访问权限
  - 类的构造函数
    - 委托构造函数
    - 拷贝构造函数
      - 拷贝构造函数的调用时机
      - 拷贝构造函数的作用
      - 重写拷贝构造函数的意义
      - 深拷贝和浅拷贝
        - 浅拷贝
      - 防止默认拷贝发生
      - 为什么参数为引用类型
    - 参数传递过程到底发生了什么？
  - 析构函数
    - 虚析构函数
- 继承
  - 公有继承（public）
  - 私有继承（private）
  - 保护继承（protected）
  - 虚继承
    - 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。
  - 继承时的名字遮蔽问题
  - C++中如何防止类被继承
- 多态和虚函数
  - 虚函数
    - 为什么有的类的析构函数需要定义成虚函数
  - 纯虚函数
  - 虚函数表

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

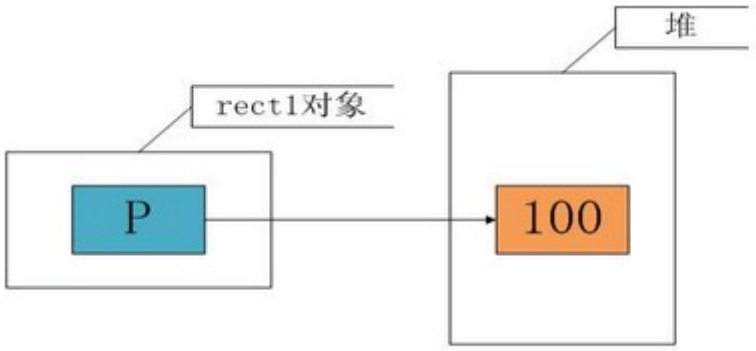
浅拷贝

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：

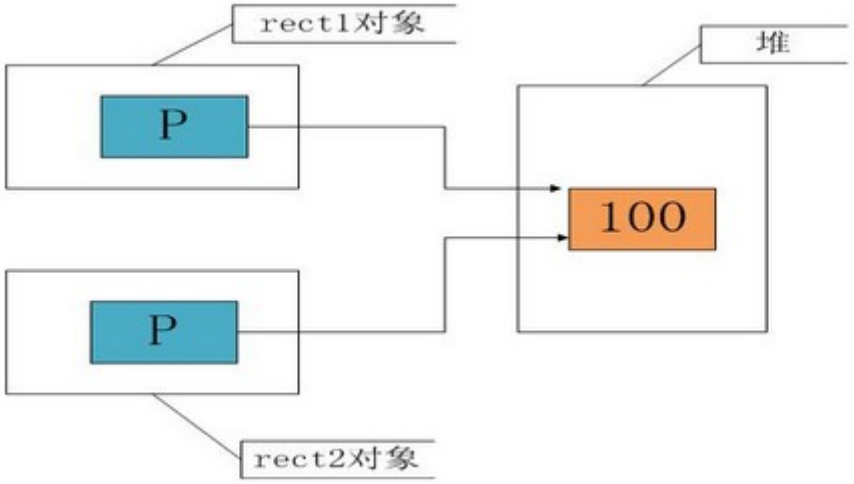
```
class Rect
{
public:
   Rect()
   {
       p=new int(100);
   }

   ~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：



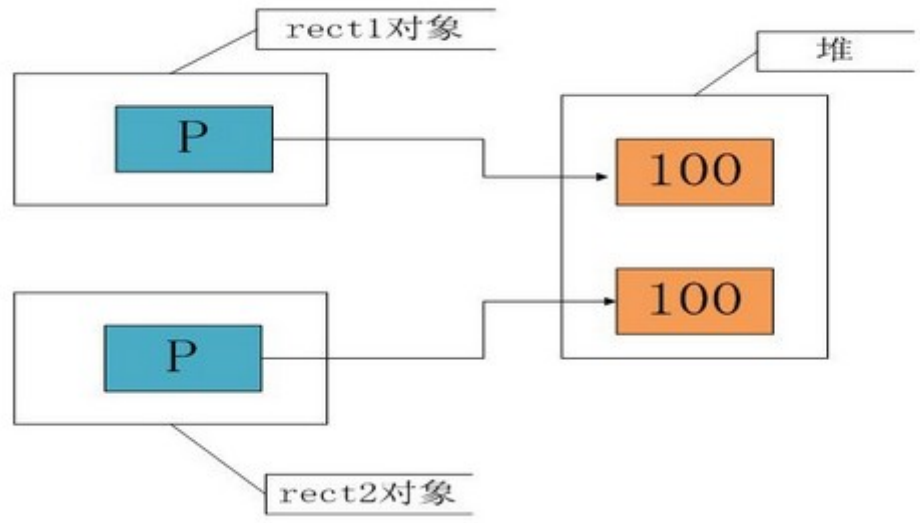
当然，这不是我们所期望的结果，在销毁对象时，两个对象的析构函数将对同一个内存空间释放两次，这就是错误出现的原因。我们需要的不是两个p有相同的值，而是两个p指向的空间有相同的值，解决办法就是使用“深拷贝”。

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。

继承

继承方式包括 public（公有的）、private（私有的）和 protected（受保护的），此项是可选的，如果不写，那么默认为 private。

```
class 派生类名:[继承方式] 基类名{
    派生类新增加的成员
};
```

派生的目的
当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。

不同继承方式的影响主要体现在：

1. 派生类成员对基类成员的访问权限；
2. 通过派生类对象对基类成员的访问权限。

公有继承（public）

它建立一种is-a的关系。 即派生类对象也是一个基类对象，可以对基类对象执行的任何操作，也可以对派生类对象执行。
继承的访问控制

1. 基类的public和protected成员：访问属性在派生类中保持不变；
2. 基类的private成员：不可直接访问。

访问权限

1. 派生类中的成员函数：可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员；
2. 通过派生类的对象：只能访问public成员。

私有继承（private）

继承的访问控制：

1. 基类的public和protected成员：都以private身份出现在派生类中；
2. 基类的private成员：不可直接访问。

访问权限：

1. 派生类中的成员函数：可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员；
2. 通过派生类的对象：不能直接访问从基类继承的任何成员。

私有继承的作用

父类的 public 和 protected 成员在子类中变成了子类 private 的成员，
这就意味着从父类继承过来的这些成员(public/protected), 子类的成员函数可以调用之;但是子类的对象就不能够调用之；
进一步的理解就是，在 子类中可以调用父类的(public/private)接口, 但是这些接口不会被暴露出去。
私有继承可以实现 has a 的关系，也就是包含。

保护继承（protected）

继承的访问控制

1. 基类的public和protected成员：都以protected身份出现在派生类中
2. 基类的private成员：不可直接访问。

访问权限

1. 派生类中的成员函数：可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员；
2. 通过派生类的对象：不能直接访问从基类继承的任何成员。

protected成员的特点与作用

1. 对建立所在类对象的模块来说，它与private成员的性质相同。
2. 对于其派生类来说，它与public成员的性质相同。
3. 既实现了数据隐藏，有方便继承，实现代码重用。

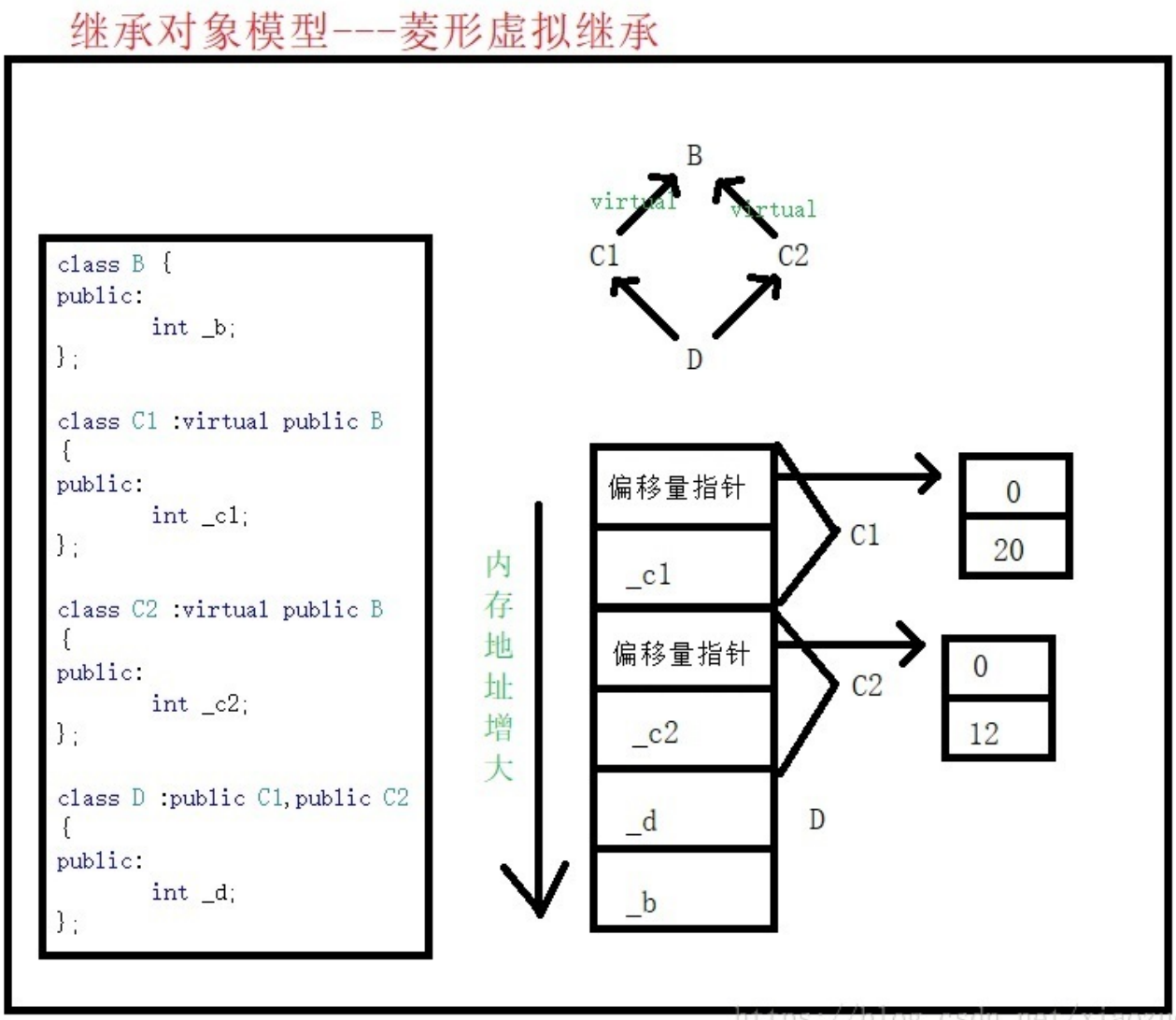
虚继承

虚继承是解决C++多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题，通常可以将派生类对象的地址赋值给基类对象，实现的具体方式是，将基类指针指向继承类（继承类有基类的拷贝）中的基类对象的地址，但是多重继承可能存在一个基类的多份拷贝，这就出现了二义性。

虚继承底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

vbptr指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

对比虚函数的实现原理：他们有相似之处，都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）。
虚基类依旧存在继承类中，只占用存储空间；虚函数不占用存储空间。
虚基类表存储的是虚基类相对直接继承类的偏移；而虚函数表存储的是虚函数地址。



```
class A
{
public:
        int dataA;
};

class B : virtual public A
{
public:
        int dataB;
};

class C : virtual public A
{
public:
        int dataC;
};

class D : public B, public C
{
public:
        int dataD;
};
```

菱形继承体系中的子类在内存布局上和普通多继承体系中的子类有很大的不一样。对于类B和C，sizeof的值变成了12，除了包含类A的成员变量dataA外还多了一个指针vbptr，类D除了继承B、C各自的成员变量dataB、dataA和自己的成员变量外，还有两个分别属于B、C的vbptr。

那么类D对象的内存布局就变成如下的样子：
vbptr：继承自父类B中的指针
int dataB：继承自父类B的成员变量
vbptr：继承自父类C的指针
int dataC：继承自父类C的成员变量
int dataD：D自己的成员变量
int A：继承自父类A的成员变量

虚继承之所以能够实现在多重派生子类中只保存一份共有基类的拷贝，关键在于vbptr指针。那vbptr到底指的是什么？又是如何实现虚继承的呢？其实上面的类D内存布局图中已经给出答案：

```
D::$vbtable@B@:
 0       | 0
 1       | 20 (Dd(B+0)A)

D::$vbtable@C@:
 0       | 0
 1       | 12 (Dd(C+0)A)
```

实际上，vbptr指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚表（virtual table），虚表中记录了vbptr与本类的偏移地址；第二项是vbptr到共有基类元素之间的偏移量。在这个例子中，类B中的vbptr指向了虚表D::$vbtable@B@，虚表表明公共基类A的成员变量dataA距离类B开始处的位移为20，这样就找到了成员变量dataA，而虚继承也不用像普通多继承那样维持着公共基类的两份同样的拷贝，节省了存储空间。

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。

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

class A
{
public:
    virtual void fun() { cout << "A::fun()." << endl; }
};

class B :public virtual A
{
public:
    void fun() { cout << "B::fun()." << endl; }
};

class C :public virtual A
{
public:
    void fun() { cout << "C::fun()." << endl; }
};

class D :public B, public C
{
public:
    void fun() { cout << "D::fun()." << endl; }
};

int main()
{
    A* a = new D;
    A* b = new B;
    a = b;
    a->fun();
    return 0;
}
```

继承时的名字遮蔽问题

如果派生类中的成员（包括成员变量和成员函数）和基类中的成员重名，那么就会遮蔽从基类继承过来的成员。所谓遮蔽，就是在派生类中使用该成员（包括在定义派生类时使用，也包括通过派生类对象访问该成员）时，实际上使用的是派生类新增的成员，而不是从基类继承来的。

基类成员函数和派生类成员函数不构成重载

基类成员和派生类成员的名字一样时会造成遮蔽，这句话对于成员变量很好理解，对于成员函数要引起注意，不管函数的参数如何，只要名字一样就会造成遮蔽。换句话说，基类成员函数和派生类成员函数不会构成重载，如果派生类有同名函数，那么就会遮蔽基类中的所有同名函数，不管它们的参数是否一样。

C++中如何防止类被继承

1. 最简单的方法就是将该类的构造函数声明为私有方法，但是这又带来另一个弊端：那就是该类本身不能生成对象了，当然这样能够满足该类不能被继承的要求，却得不偿失。
2. 将A类虚继承E类，但是E类的构造函数是带private属性的，A类还是E类的友元。

   分析：

   如果我们让A类虚继承E类，根据虚继承的特性，虚基类的构造函数由最终的子类负责构造，此时E类的构造函数虽然是私有的，但是A类是E类的友元，所以可以调用E类的构造函数完成初始化。

   B类如果要想继承A类，它必须能够调用E虚基类的构造函数来完成初始化，这是不可能的，因为它不是E类的友元！因此，我们的A类也就终于成为了一个无法继承的类，并且我们还可以在A类外实例化对象，可以正常使用。

```
class E
{
private:
    friend class A;
    E(){}
};
class A : virtual public E
{
public:
    A(){}
};
```

3. final关键字

   final关键字用于虚函数时可以防止虚函数被子类重写，用于类时可以防止类被继承。

```
//类A可以被实例化，无法被继承
class A final {
public:
    A(){}
};
```

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。

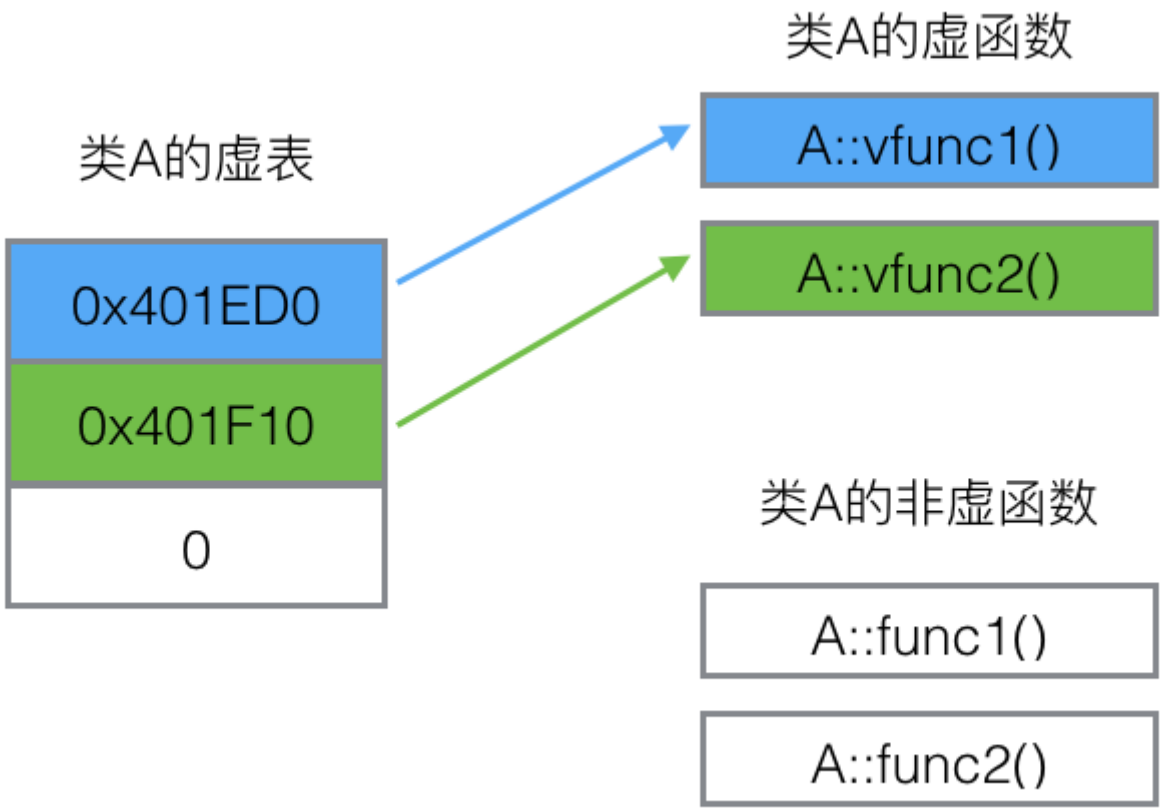
虚函数表

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;
};
```

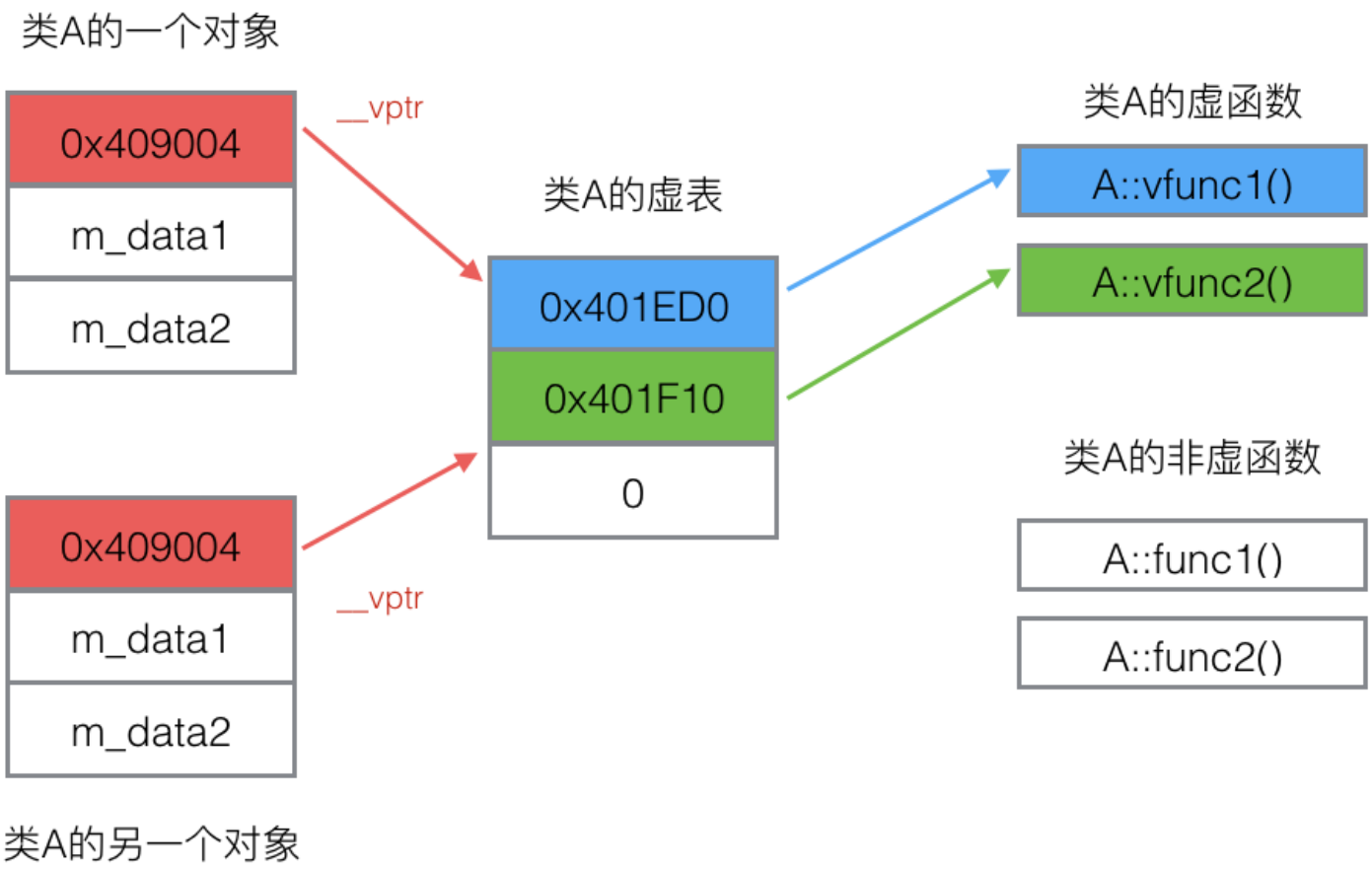


虚表是一个指针数组，其元素是虚函数的指针，每个元素对应一个虚函数的函数指针。需要指出的是，普通的函数即非虚函数，其调用并不需要经过虚表，所以虚表的元素并不包括普通函数的函数指针。
虚表内的条目，即虚函数指针的赋值发生在编译器的编译阶段，也就是说在代码的编译阶段，虚表就可以构造出来了。

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。

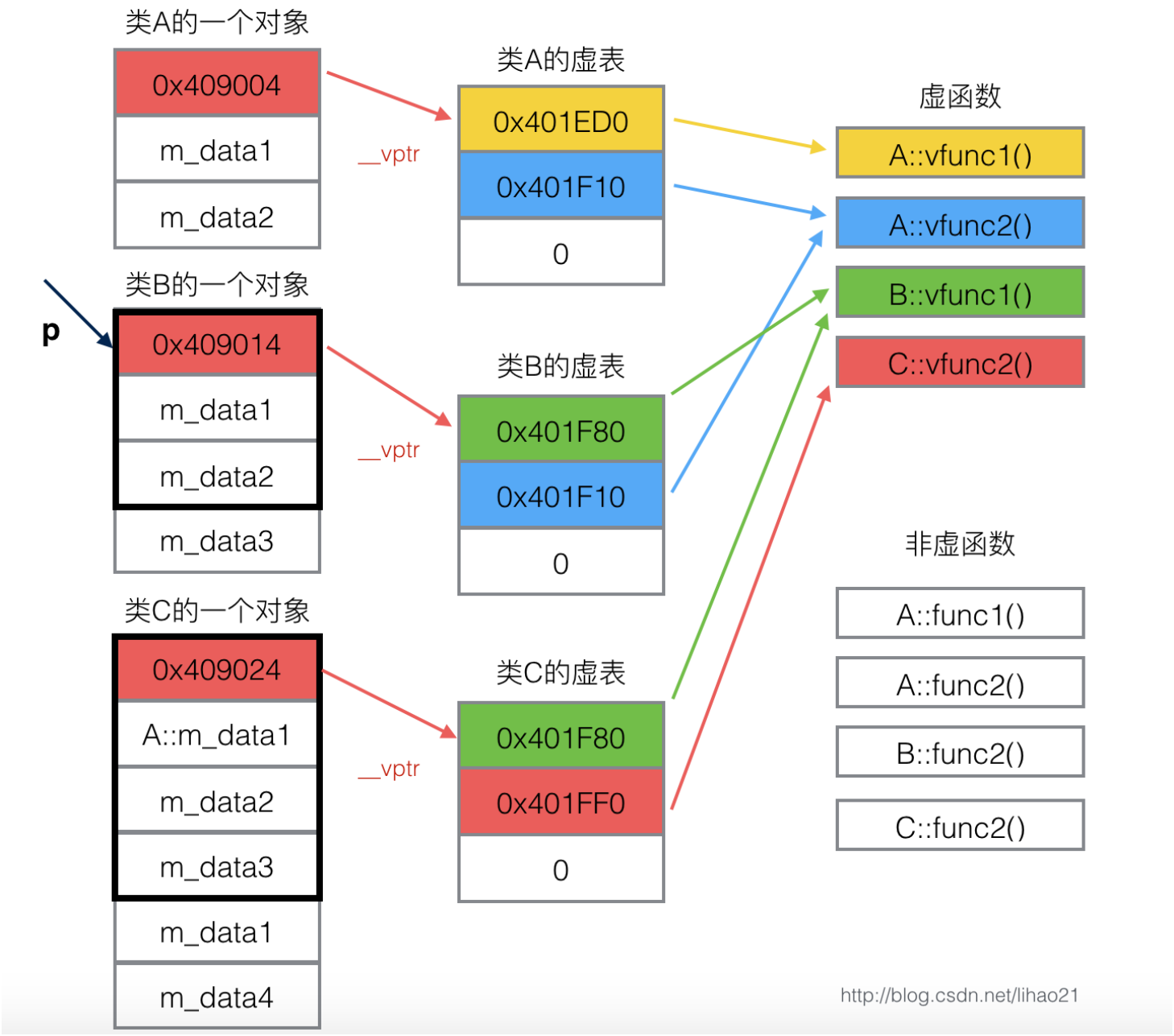


动态绑定

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();
}
```

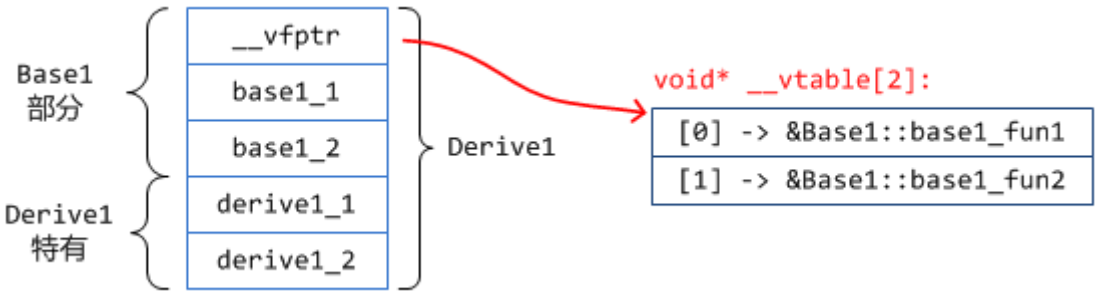
当aObject在创建时，它的虚表指针__vptr已设置为指向A vtbl，这样p->__vptr就指向A vtbl。vfunc1在A vtbl对应在条目指向了A::vfunc1()函数，所以 p->vfunc1()实质会调用A::vfunc1()函数。

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;
};
```

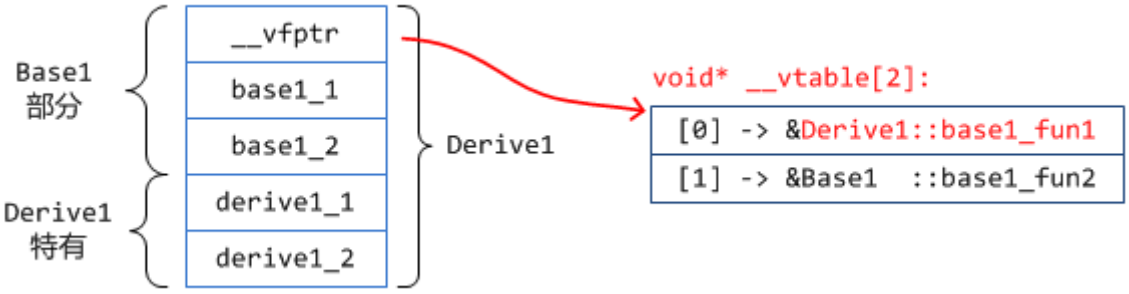
现在类的布局情况应该是下面这样:



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() {}
};
```

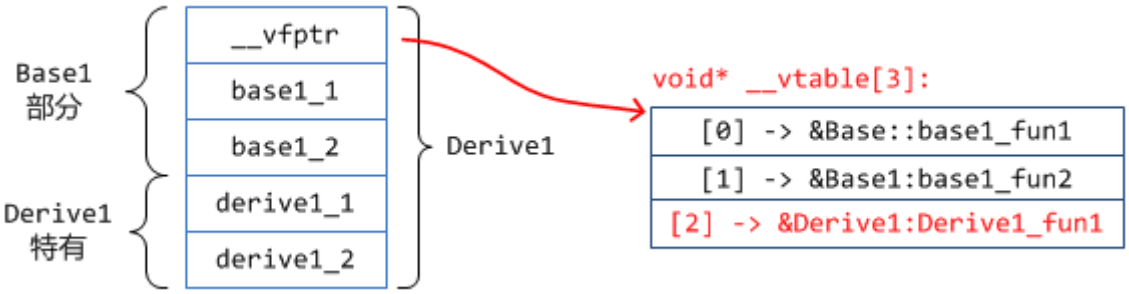
Derive1类 重写了Base1类的base1_fun1()函数, 也就是常说的虚函数覆盖。无论是通过Derive1的指针还是Base1的指针来调用此方法, 调用的都将是被继承类重写后的那个方法(函数), 这时就产生了多态。那么新的布局图:



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() {}
};
```

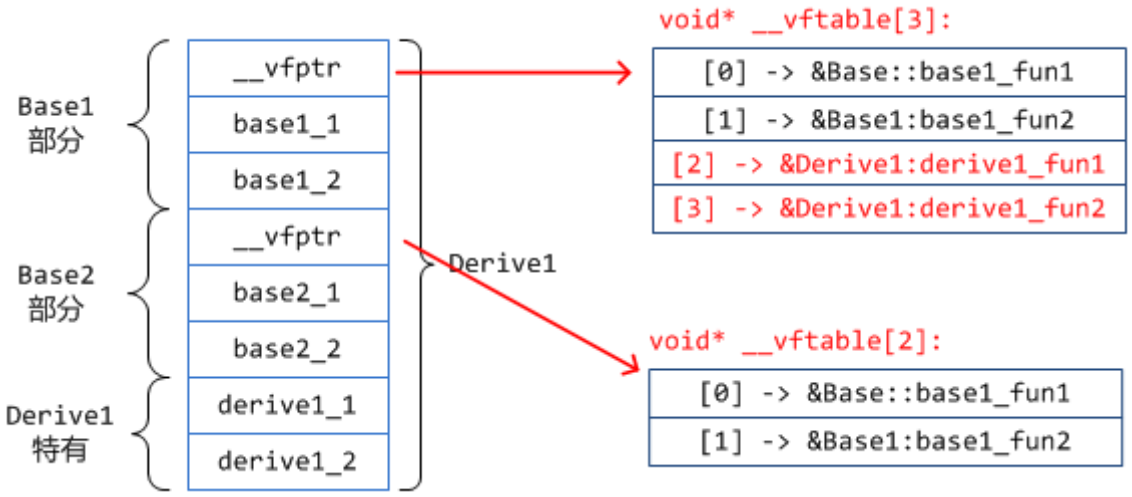
继承类Derive1的虚函数表被加在基类的后面。
由于Base1只知道自己的两个虚函数索引[0][1], 所以就算在后面加上了[2], Base1根本不知情, 不会对她造成任何影响。



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() {}
};
```

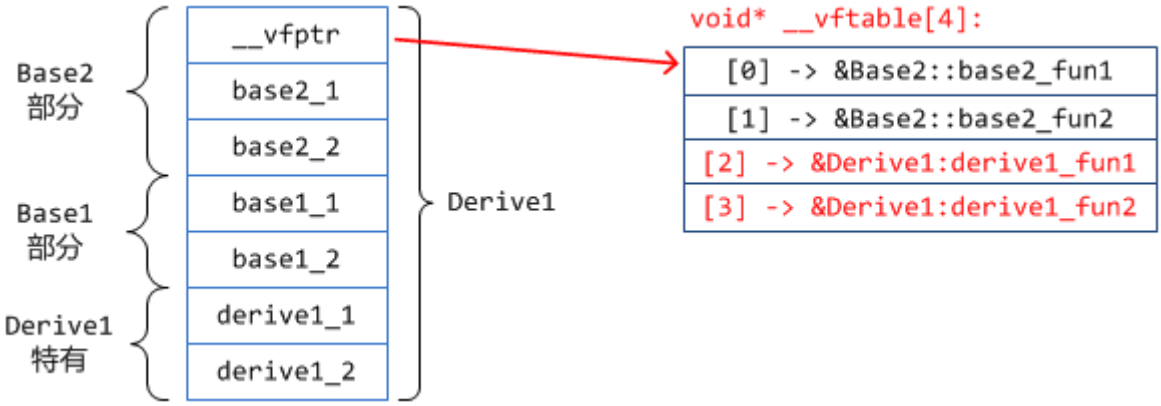
Derive1的虚函数表依然是保存到第1个拥有虚函数表的那个基类的后面的.
看看现在的类对象布局图:



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() {}
};
```