多态

通过基类指针只能访问派生类的成员变量，但是不能访问派生类的成员函数。

为了消除这种尴尬，让基类指针能够访问派生类的成员函数，C++ 增加了**虚函数（Virtual Function）**。使用虚函数非常简单，只需要在函数声明前面增加 virtual 关键字。

有了虚函数，基类指针指向基类对象时就使用基类的成员（包括成员函数和成员变量），指向派生类对象时就使用派生类的成员。换句话说，基类指针可以按照基类的方式来做事，也可以按照派生类的方式来做事，它有多种形态，或者说有多种表现方式，我们将这种现象称为**多态（Polymorphism）**。

C++提供多态的目的是：可以通过基类指针对所有派生类（包括直接派生和间接派生）的成员变量和成员函数进行“全方位”的访问，尤其是成员函数。如果没有多态，我们只能访问成员变量。（与 https://c.biancheng.net/view/2284.html 向上转型对比，如果没有用虚函数构成多态，则：编译器通过指针来访问成员变量，指针指向哪个对象就使用哪个对象的数据；编译器通过指针的类型来访问成员函数，指针属于哪个类的类型就使用哪个类的函数。）

通过指针调用普通的成员函数时会根据指针的类型（通过哪个类定义的指针）来判断调用哪个类的成员函数，但是通过本节的分析可以发现，这种说法并不适用于虚函数，虚函数是根据指针的指向来调用的，指针指向哪个类的对象就调用哪个类的虚函数。（这根成员变量一样，成员变量无论有无虚函数都是根据指针的指向来访问的）

【例子】

1. 没有用虚函数

```
//基类People
class People{
public:
    People(char *name, int age);
    void display();
protected:
    char *m_name;
    int m_age;
};
People::People(char *name, int age): m_name(name), m_age(age){}
void People::display(){
    cout<<m_name<<"今年"<<m_age<<"岁了，是个无业游民。"<<endl;
}

//派生类Teacher
class Teacher: public People{
public:
    Teacher(char *name, int age, int salary);
    void display();
private:
    int m_salary;
};
Teacher::Teacher(char *name, int age, int salary): People(name, age), m_salary(salary){}
void Teacher::display(){
    cout<<m_name<<"今年"<<m_age<<"岁了，是一名教师，每月有"<<m_salary<<"元的收入。"<<endl;
}

int main(){
    People *p = new People("王志刚", 23);
    p -> display();

    p = new Teacher("赵宏佳", 45, 8200);
    p -> display();

    return 0;
}
```

运行结果：
王志刚今年23岁了，是个无业游民。
赵宏佳今年45岁了，是个无业游民。

解释：p 定义的时候是 People 类的指针，因此无论它指向哪个类的对象，都只会调用 People 类的成员函数

2. 使用了虚函数

```
//基类People
class People{
public:
    People(char *name, int age);
    virtual void display();  //声明为虚函数
protected:
    char *m_name;
    int m_age;
};
People::People(char *name, int age): m_name(name), m_age(age) {}
void People::display() {
    cout<<m_name<<"今年"<<m_age<<"岁了，是个无业游民。"<<endl;
}

//派生类Teacher
class Teacher: public People{
public:
    Teacher(char *name, int age, int salary);
    virtual void display();  //声明为虚函数
private:
    int m_salary;
};
Teacher::Teacher(char *name, int age, int salary): People(name, age), m_salary(salary) {}
void Teacher::display() {
    cout<<m_name<<"今年"<<m_age<<"岁了，是一名教师，每月有"<<m_salary<<"元的收入。"<<endl;
}

int main() {
    People *p = new People("王志刚", 23);
    p -> display();

    p = new Teacher("赵宏佳", 45, 8200);
    p -> display();

    return 0;
}
```

运行结果：
王志刚今年23岁了，是个无业游民。
赵宏佳今年45岁了，是一名教师，每月有8200元的收入。

解释：虽然 p 定义是 People 类的指针，但由于有了虚函数，p 指向哪个类的对象，就会调用这个类的成员函数，所以第二次 p 指向了 Teacher 类的对象，调用的就是 Teacher 的成员函数。

借助引用也可以实现多态

引用在本质上是通过指针的方式实现的，这一点已在《C++引用在本质上是什么，它和指针到底有什么区别？》中进行了讲解，既然借助指针可以实现多态，那么我们就有理由推断：**借助引用也可以实现多态。**

```
01. int main(){
02.     People p("王志刚", 23);
03.     Teacher t("赵宏佳", 45, 8200);
04. 
05.     People &rp = p;
06.     People &rt = t;
07. 
08.     rp.display();
09.     rt.display();
10. 
11.     return 0;
12. }
```

运行结果：
王志刚今年23岁了，是个无业游民。
赵宏佳今年45岁了，是一名教师，每月有8200元的收入。

解释：当基类的引用指代基类对象时，调用的是基类的成员，而指代派生类对象时，调用的是派生类的成员。

不过引用不像指针灵活，指针可以随时改变指向，而引用只能指代固定的对象，在多态性方面缺乏表现力，所以以后我们再谈及多态时一般是说指针。

构成多态的条件

C++ 虚函数对于多态具有决定性的作用，有虚函数才能构成多态。上节《C++多态和虚函数快速入门教程》我们已经介绍了虚函数的概念，这节我们来重点说一下虚函数的注意事项。

1) 只需要在虚函数的声明处加上 virtual 关键字，函数定义处可以加也可以不加。

2) 为了方便，你可以只将基类中的函数声明为虚函数，这样所有派生类中具有遮蔽关系的同名函数都将自动成为虚函数。关于名字遮蔽已在《C++继承时的名字遮蔽》一节中进行了讲解。

3) 当在基类中定义了虚函数时，如果派生类没有定义新的函数来遮蔽此函数，那么将使用基类的虚函数。

4) 只有派生类的虚函数覆盖基类的虚函数（函数原型相同）才能构成多态（通过基类指针访问派生类函数）。例如基类虚函数的原型为 `virtual void func();`，派生类虚函数的原型为 `virtual void func(int);`，那么当基类指针 p 指向派生类对象时，语句 `p -> func(100);` 将会出错，而语句 `p -> func();` 将调用基类的函数。

5) 构造函数不能是虚函数。对于基类的构造函数，它仅仅是在派生类构造函数中被调用，这种机制不同于继承。也就是说，派生类不继承基类的构造函数，将构造函数声明为虚函数没有什么意义。

6) 析构函数可以声明为虚函数，而且有时候必须要声明为虚函数，这点我们将在下节中讲解。

https://c.biancheng.net/view/2296.html

下面是构成多态的条件：

- 必须存在继承关系；
- 继承关系中必须有同名的虚函数，并且它们是覆盖关系（函数原型相同）。
- 存在基类的指针，通过该指针调用虚函数。

函数原型相同指的是函数名和形参列表都相同，多态特征在于：基类指针如果指向基类对象，就可以访问基类成员变量，调用基类成员函数；如果基类指针指向派生类对象，就可以访问派生类成员变量，调用派生类成员函数(如果没有虚函数构成多态的话，基类指针如果指向派生类对象，那么只能访问派生类的成员变量，即使有同名函数构成遮蔽，调用成员函数时还是只能调用基类自己的成员函数)

(函数名字遮蔽现象只是针对于派生类对象来说的（多态针对的是基类指针来说的)，就是派生类有和基类同名的函数（形参列表不一定要相同，同名就行)，那么派生类对象在调用该函数时调用的是派生类中定义的该函数，而不是基类中与之同名的函数)

什么时候声明虚函数

首先看成员函数所在的类是否会作为基类。然后看成员函数在类的继承后有无可能被更改功能，如果希望更改其功能的，一般应该将它声明为虚函数。如果成员函数在类被继承后功能不需修改，或派生类用不到该函数，则不要把它声明为虚函数。

纯文本 复制

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

//基类Base
class Base{
public:
    virtual void func();
    virtual void func(int);
};
void Base::func(){
    cout<<"void Base::func()"<<endl;
}
void Base::func(int n){
    cout<<"void Base::func(int)"<<endl;
}

//派生类Derived
class Derived: public Base{
public:
    void func();
    void func(char *);
};
void Derived::func(){
    cout<<"void Derived::func()"<<endl;
}
void Derived::func(char *str){
    cout<<"void Derived::func(char *)"<<endl;
}

int main(){
    Base *p = new Derived();
    p -> func();  //输出void Derived::func()
    p -> func(10);  //输出void Base::func(int)
    p -> func("http://c.biancheng.net");  //compile error
```

注意：最后一行报错是因为 func(char* str)虽然与基类的一个函数 func 名字相同但形参列表不同，也就是说函数原型不同，因此不构成多态，所以通过基类的指针只能调用基类的成员函数，而基类没有 func(char* str)这样的成员函数，因此编译错误。

纯虚函数

在C++中，可以将虚函数声明为纯虚函数，语法格式为：

```
virtual 返回值类型 函数名 (函数参数) = 0;
```

纯虚函数没有函数体，只有函数声明，在虚函数声明的结尾加上 `=0` ，表明此函数为纯虚函数。

> 最后的 `=0` 并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是纯虚函数”。

包含纯虚函数的类称为抽象类（Abstract Class）。之所以说它抽象，是因为它无法实例化，也就是无法创建对象。原因很明显，纯虚函数没有函数体，不是完整的函数，无法调用，也无法为其分配内存空间。

抽象类通常是作为基类，让派生类去实现纯虚函数。派生类必须实现纯虚函数才能被实例化。

关于纯虚函数的几点说明

1) 一个纯虚函数就可以使类成为抽象基类，但是抽象基类中除了包含纯虚函数外，还可以包含其它的成员函数（虚函数或普通函数）和成员变量。

2) 只有类中的虚函数才能被声明为纯虚函数，普通成员函数和顶层函数均不能声明为纯虚函数。如下例所示：

纯文本 复制

```
//顶层函数不能被声明为纯虚函数
void fun() = 0;   //compile error

class base{
public :
    //普通成员函数不能被声明为纯虚函数
    void display() = 0;  //compile error
};
```

继承

2. 继承中的名字遮蔽现象：

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。

链接：https://c.biancheng.net/view/2271.html

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。

链接：https://c.biancheng.net/view/2271.html

3. 继承方式：

> ## public、protected、private 指定继承方式
>
> 不同的继承方式会影响基类成员在派生类中的访问权限。
>
> **1) public继承方式**
> - 基类中所有 public 成员在派生类中为 public 属性；
> - 基类中所有 protected 成员在派生类中为 protected 属性；
> - 基类中所有 private 成员在派生类中不能使用。
>
> **2) protected继承方式**
> - 基类中的所有 public 成员在派生类中为 protected 属性；
> - 基类中的所有 protected 成员在派生类中为 protected 属性；
> - 基类中的所有 private 成员在派生类中不能使用。
>
> **3) private继承方式**
> - 基类中的所有 public 成员在派生类中均为 private 属性；
> - 基类中的所有 protected 成员在派生类中均为 private 属性；
> - 基类中的所有 private 成员在派生类中不能使用。

下表汇总了不同继承方式对不同属性的成员的影响结果

继承方式/基类成员	public成员	protected成员	private成员
public继承	public	protected	不可见
protected继承	protected	protected	不可见
private继承	private	private	不可见

4. 多继承

多继承的语法也很简单，将多个基类用逗号隔开即可。例如已声明了类A、类B和类C，那么可以这样来声明派生类D：

```
class D: public A, private B, protected C{
    //类D新增加的成员
}
```

D 是多继承形式的派生类，它以公有的方式继承 A 类，以私有的方式继承 B 类，以保护的方式继承 C 类。D 根据不同的继承方式获取 A、B、C 中的成员，确定它们在派生类中的访问权限。

多继承下的构造函数

多继承形式下的构造函数和单继承形式基本相同，只是要在派生类的构造函数中调用多个基类的构造函数。以上面的 A、B、C、D 类为例，D 类构造函数的写法为：

```
D(形参列表): A(实参列表), B(实参列表), C(实参列表){
    //其他操作
}
```

基类构造函数的调用顺序和和它们在派生类构造函数中出现的顺序无关，而是和声明派生类时基类出现的顺序相同。仍然以上面的 A、B、C、D 类为例，即使将 D 类构造函数写作下面的形式：

```
D(形参列表): B(实参列表), C(实参列表), A(实参列表){
    //其他操作
}
```

那么也是先调用 A 类的构造函数，再调用 B 类构造函数，最后调用 C 类构造函数。

注意：多个基类的构造函数调用顺序之和继承时的顺序有关，即 class D : public A, private B, protected C，则调用顺序为 A,B,C，跟子类 D 构造函数中初始化它的基类 A,B,C 的先后顺序无关。

5. 虚继承与虚基类

https://c.biancheng.net/view/2280.html

https://c.biancheng.net/view/2281.html

应用：解决菱形继承间接基类（爷爷类）的命名冲突问题

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。其中，这个被共享的基类就称为虚基类（Virtual Base Class），本例中的 A 就是一个虚基类。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含一份虚基类的成员。

现在让我们重新梳理一下本例的继承关系，如下图所示：

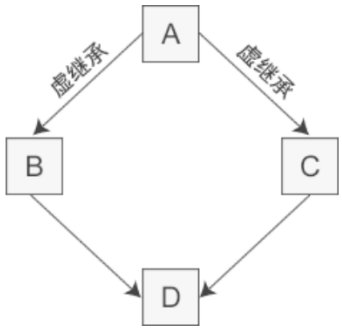


图2：使用虚继承解决菱形继承中的命名冲突问题

为了避免出现这种矛盾的情况，C++ 干脆规定必须由最终的派生类 D 来初始化虚基类 A，直接派生类 B 和 C 对 A 的构造函数的调用是无效的。在第 50 行代码中，调用 B 的构造函数时试图将 m_a 初始化为 90，调用 C 的构造函数时试图将 m_a 初始化为 100，但是输出结果有力地证明了这些都是无效的，m_a 最终被初始化为 50，这正是在 D 中直接调用 A 的构造函数的结果。

另外需要关注的是构造函数的执行顺序。虚继承时构造函数的执行顺序与普通继承时不同：在最终派生类的构造函数调用列表中，不管各个构造函数出现的顺序如何，编译器总是先调用虚基类的构造函数，再按照出现的顺序调用其他的构造函数；而对于普通继承，就是按照构造函数出现的顺序依次调用的。

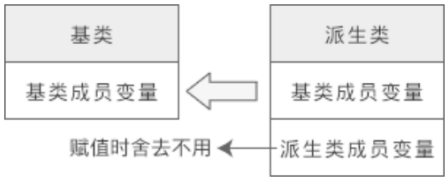
在虚继承中，虚基类是由最终的派生类初始化的，换句话说，最终派生类的构造函数必须要调用虚基类的构造函数。对最终的派生类来说，虚基类是间接基类，而不是直接基类。这跟普通继承不同，在普通继承中，派生类构造函数中只能调用直接基类的构造函数，不能调用间接基类的。

6. 派生类对象赋值给基类对象

https://c.biancheng.net/view/2284.html

赋值的本质是将现有的数据写入已分配好的内存中，对象的内存只包含了成员变量，所以对象之间的赋值是成员变量的赋值，成员函数不存在赋值问题。运行结果也有力地证明了这一点，虽然有 a=b; 这样的赋值过程，但是 a.display() 始终调用的都是 A 类的 display() 函数。换句话说，对象之间的赋值不会影响成员函数，也不会影响 this 指针。

将派生类对象赋值给基类对象时，会舍弃派生类新增的成员，也就是“大材小用”，如下图所示：



可以发现，即使将派生类对象赋值给基类对象，基类对象也不会包含派生类的成员，所以依然不同通过基类对象来访问派生类的成员。对于上面的例子，a.m_a 是正确的，但 a.m_b 就是错误的，因为 a 不包含成员 m_b。

这种转换关系是不可逆的，只能用派生类对象给基类对象赋值，而不能用基类对象给派生类对象赋值。理由很简单，基类不包含派生类的成员变量，无法对派生类的成员变量赋值。同理，同一基类的不同派生类对象之间也不能赋值。

将派生类指针赋值给基类指针

编译器虽然通过指针的指向来访问成员变量，但是却不通过指针的指向来访问成员函数：编译器通过指针的类型来访问成员函数。对于 pa，它的类型是 A，不管它指向哪个对象，使用的都是 A 类的成员函数，具体原因已在《C++函数编译原理和成员函数的实现》中做了详细讲解。

概括起来说就是：编译器通过指针来访问成员变量，指针指向哪个对象就使用哪个对象的数据；编译器通过指针的类型来访问成员函数，指针属于哪个类的类型就使用哪个类的函数。

将派生类引用赋值给基类引用

引用在本质上是通过指针的方式实现的，这一点已在《引用在本质上是什么，它和指针到底有什么区别》中进行了讲解，既然基类的指针可以指向派生类的对象，那么我们就有理由推断：基类的引用也可以指向派生类的对象，并且它的表现和指针是类似的。

引用和指针的表现之所以如此类似，是因为引用和指针并没有本质上的区别，引用仅仅是对指针进行了简单封装，读者可以猛击《引用在本质上是什么，它和指针到底有什么区别》一文深入了解。

引用和指针的关系与区别：

引用变量在功能上等于一个指针常量，即一旦指向某一个单元就不能再指向别处。在底层，引用变量由指针按照指针常量的方式实现。

指针比较灵活，可以自增自减，可以指向不同的地址，引用则是指针常量，一旦初始化后就不能改变指向的地址。

```
int i = 5;
```

引用：`int &ri = i;`

等价于：指针常量：`int* const pi = &i;`