C++虚函数

C++虚函数是多态性实现的重要方式，当某个虚函数通过指针或者引用调用时，编译器产生的代码直到运行时才能确定到底调用哪个版本的函数。被调用的函数是与绑定到指针或者引用上的对象的动态类型相匹配的那个。因此，借助虚函数，我们可以实现多态性。这也是OOP的核心思想之一。

引言

考虑下面一个继承的例子，Dog类与Cat类都继承自Animal类，但是它们拥有不同的speak()方法：

```
class Animal
{
public:
    Animal(const string& name):
        m_name{name}
    {}

    const string& getName() const
    {
        return m_name;
    }

    string speak() const
    {
        return "???";
    }

private:
    string m_name;
};

class Cat : public Animal
{
public:
    Cat(const string& name):
        Animal(name)
    {}

    string speak() const
    {
        return "Meow";
    }
};

class Dog : public Animal
{
```

```
public:
    Dog(const string& name):
        Animal(name)
    {}

    string speak() const
    {
        return "Woof";
    }
};
```

我们知道派生类对象可以赋值给基类的指针或者引用，但是我们希望调用这些指针或者引用时，能够调用各个派生类自己的方法，比如下面的例子：

```
int main()
{
    Cat cat{ "Fred" };
    cout << "Cat is named " << cat.getName() << ", and it says " << cat.speak() << endl;

    Dog dog{ "Carbo" };
    cout << "Dog is named " << dog.getName() << ", and it says " << dog.speak() << endl;

    Animal* catAnimal = &cat;
    cout << "Cat is named " << catAnimal->getName() << ", and it says " << catAnimal->speak() << endl;

    Animal& dogAnimal = dog;
    cout << "Dog is named " << dogAnimal.getName() << ", and it says " << dogAnimal.speak() << endl;

        return 0;
}
```

但是输出并不是预期的那样：

```
cat is named Fred, and it says Meow
dog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

无论是指针还是引用，它们都没有调用其派生对象所重写的方法，而是基类原有的方法。大家可能会想，为什么我非要将派生类对象赋值给基类的指针或者引用来调用派生类的方法？直接利用派生类对象调用不就可以了吗？这样做有很多好处，比如你想使用一个函数，接收一个动物对象类，然后打印其名字以及叫声。但是由于这样的动物类有两个，你必须利用重载的思想实现两个版本：

```
void print(Cat& cat)
{
```

```
    cout << cat.getName() << " says " << cat.speak() << endl;
}

void print(Dog& dog)
{
    cout << dog.getName() << " says " << dog.speak() << endl;
}
```

两个版本实现起来并没有那么麻烦，但是如果动物类的种类更多呢？这个时候你就有点不乐意了，仅仅是对象类型不同，但是方法是相同的，为什么不能仅写一个版本：

```
void print(Animal& animal)
{
    cout << animal.getName() << " says " << animal.speak() << endl;
}
```

如果基类能够动态确定其实际所指向的派生类对象，并调用合适版本的方法，那么一个函数就可以解决上面的问题。

看来尽管每个派生类都有自己实现的 speak()方法，但是它们实际上并没有真正的重写基类方法，仅仅是隐藏。因为派生类对象传递给基类的指针或者引用并没有调用派生类版本的方法，依然是基类方法。

所以，你需要虚函数！

虚函数与多态性

虚函数是类方法中的一种特殊函数，当你调用它时，它会匹配派生最远的重写版本。这种特性是多态性。匹配的规则是相同的函数签名（函数名，参数个数与类型）以及返回类型（返回类型可以不相同，但必须存在派生关系）。虚函数仅需要再前面加上一个 virtual 关键字即可，利用虚函数我们可以修改上面的代码：

```
class Animal
{
public:
    Animal(const string& name):
        m_name{name}
    {}

    const string& getName() const
    {
        return m_name;
    }

    virtual string speak() const
    {
        return "???";
    }

private:
    string m_name;
};
```

```
class Cat : public Animal
{
public:
    Cat(const string& name):
        Animal(name)
    {}

    virtual string speak() const
    {
        return "Meow";
    }
};

class Dog : public Animal
{
public:
    Dog(const string& name):
        Animal(name)
    {}

    virtual string speak() const
    {
        return "Woof";
    }
};
```

此时，再测试一下下面的代码，可以看到输出实现了预期的效果：

```
int main()
{
    Cat cat{ "Fred" };
    cout << "Cat is named " << cat.getName() << ", and it says " << cat.speak() <<
endl;

    Dog dog{ "Carbo" };
    cout << "Dog is named " << dog.getName() << ", and it says " << dog.speak() <<
endl;

    Animal* catAnimal = &cat;
    cout << "Cat is named " << catAnimal->getName() << ", and it says " <<
catAnimal->speak() << endl;

    Animal& dogAnimal = dog;
    cout << "Dog is named " << dogAnimal.getName() << ", and it says " <<
dogAnimal.speak() << endl;
```

```
        return 0;
}
output:
cat is named Fred, and it says Meow
dog is named Garbo, and it says Woof
cat is named Fred, and it says Meow
dog is named Garbo, and it says Woof
```

可以看到，不论是基类版本还是派生类版本，我们都在函数前面使用了 virtual 关键字，事实上，派生类中的 virtual 关键字并不是必要的。一旦基类中的方法打上了 virtual 标签，那么派生类中匹配的函数也是虚函数。但是，还是建议在后面的派生类中加上 virtual 关键字，作为虚函数的一种提醒，以便后面可能还会有更远的派生。

注意千万不要在构造函数与析构函数中调用虚函数。我们知道派生类对象在创建时，首先基类部分先被创建，如果你在基类构造函数调用虚函数时，它此时将无法调用派生类版本的函数，因为派生类对象还未创建，此时派生类虚函数没有作用的对象。那么，它只能调用基类版本的虚函数。对于析构函数，派生类对象中的派生部分先被析构，如果你在基类析构函数中调用了虚函数，它也只能调用基类版本的虚函数，因为派生类对象已经不存在了。

到底什么时候使用虚函数？大部分时候，我们希望派生类是真正的“重写”基类函数，而不是“隐藏”。所以一般建议将所有方法都声明为 virtual。既然如此，为什么编译器不默认这样做呢，其实对于 Java 语言来说，所有的方法默认是虚函数。但是使用虚函数是有代价的，相对于普通函数，虚函数的调用代价稍高，但是这种差别不会太大，所以还是建议所有方法都使用 virtual 关键字。

override 标识符

前面说到，派生类的重写方法必须与基类方法要匹配，否则编译器会认为派生类创建了一个新方法，而不是重写基类的版本，看下面的例子：

```
class Super
{
public:
    virtual string getName1(int x)
    {
        return "Super";
    }

    virtual string getName2(int x)
    {
        return "Super";
    }
};

class Sub: public Super
{
public:
    virtual string getName1(double x)
    {
```

```
        return "Sub";
    }

    virtual string getName2(int x) const
    {
        return "Sub";
    }
};

int main()
{
    Sub sub;
    Super* super = &sub;

    cout << super->getName1(1) << endl;  // output: Super
    cout << super->getName2(2) << endl;  // output: Super

    cin.ignore(10);
    return 0;
}
```

可以看到，派生类的两个虚方法并没有重写基类版本，这是由于两个方法的函数签名并不一样。所以将派生类对象赋值给基类的指针只能是调用基类方法。但是，实际上我们希望派生类的两个方法是重写版本。有时候，我们很容易犯一些小错误导致重写失败，比如上面的例子。还有时候，我们修改了基类虚函数，但是没有更新派生类的对应重载版本，也将有可能使重写失效。为了避免这样的错误，C++引入了 override 标识符，使用这个标识符告诉编译器这是重写的方法，如果方法不匹配，那么将无法通过编译。用 override 修改代码如下：

```
class Super
{
public:
    virtual string getName1(int x)
    {
        return "Super";
    }

    virtual string getName2(int x)
    {
        return "Super";
    }
};

class Sub: public Super
{
public:
    virtual string getName1(double x) override
```

```
    {
        return "Sub";
    }

    virtual string getName2(int x) const override
    {
        return "Sub";
    }
    // 此时无法编译
};
```

所以，只要重写基类方法，建议使用 override 标识符，避免无意的错误。

final 标识符

有时候，你不想派生类重写基类的虚方法，此时可以使用 final 标识符，这个时候如果派生类重写了基类虚方法，那么将无法编译：

```
class A
{
public:
    virtual void someMethod() { cout << "A" << endl; }
}

class B: public A
{
public:
    // 基类 A 的 someMethod 方法没有 final 标识符，那么 B 可以重写该方法
    // 但是此虚方法使用了 final 标识符，后面的派生类无法重写
    virtual void someMethod() override final { cout << "B" << endl; }
}

class C: public B
{
public:
    // 无法编译，因为不允许重写
    virtual void someMethod() override { cout << "C" << endl; }
}
```

final 标识符还可以直接用于类，此时该类将不能被继承：

```
class A
{
public:
    virtual void someMethod() { cout << "A" << endl; }
};

// B 可以继承 A
class B final: public A
{
```

```
public:
    virtual void someMethod() override { cout << "B" << endl; }
};

// B 无法被继承，此时无法编译
class C: public B
{
public:
    virtual void someMethod() override { cout << "C" << endl; }
};
```

协变返回类型

前面说过，要想成功重写方法，基类虚方法与派生类虚方法必须匹配，其中返回类型也必须一致。但是有时候返回类型不相同，也能实现重写，此时返回类型存在继承关系：基类方法返回类型是一个指向某一类的指针或者引用，而派生类重写版本的返回类型是指向派生类的指针或者引用。这种情况称为协变返回类型。下面是一个例子：

```
class Super
{
public:
    virtual Super* getThis() { return this; }
};

class Sub : public Super
{
    virtual Sub* getThis() override { return this; }
};
```

析构函数要声明为虚函数

对于析构函数，大部分时间我们只需要使用编译器提供的默认版本就好，除非涉及到释放动态分配的内存。但是如果存在继承，虚函数最好声明为虚函数。否则删除一个实际指向派生类的基类指针，只会调用基类的析构函数，而不会调用派生类的析构函数以及派生类数据成员的析构函数。这样就可能造成内存泄露，看下面的例子：

```
class Resource
{
public:
    Resource() { cout << "Resource created!" << endl; }
    ~Resource() { cout << "Resource destoryed!" << endl; }
};

class Super
{
public:
    Super() { cout << "Super constructor called!" << endl; }
    ~Super() { cout << "Super destructor called!" << endl; }
};
```

```
class Sub : public Super
{
public:
    Sub() { cout << "Sub constructor called!" << endl;}

    ~Sub() { cout << "Sub destructor called!" << endl; }
private:
    Resource res;
};
```

如果执行下面的代码：

```
int main()
{
    Sub* sub = new Sub;
    Super* super = sub;
    delete super;

    cin.ignore(10);
        return 0;
}
```

其输出为：

```
Super constructor called!
Resource created!
Sub constructor called!
Super destructor called!
```

可以看到，派生类的析构函数没有执行，其数据成员 Resource 也没有被析构。但是如果你将析构函数都声明为虚函数，上面的代码将得到如下的结果：

```
Super constructor called!
Resource created!
Sub constructor called!
Resource destoryed!
Sub destructor called!
Super destructor called!
```

此时，程序按照预期输出，所以，对于继承问题，没有理由不将析构函数声明为虚函数！

函数调用捆绑

要想深刻理解虚函数机理，首先要了解函数调用捆绑机制。捆绑指的是将标识符（如变量名与函数名）转化为地址。这里我们仅仅关注有关函数调用的捆绑。我们知道每个函数在编译的过程中是存在一个唯一的地址的。如果我们在程序段里面直接调用某个函数，那么编译器或者链接器会直接将函数标识符替换为一个机器地址。这种方式是早捆绑，或者说是静态捆绑。因为捆绑是在程序运行之前完成的。看下面的简单例子：

```
int add(int x, int y)
{
    return x + y;
}
```

```
int subtract(int x, int y)
{
    return x - y;
}

int multiply(int x, int y)
{
    return x * y;
}

int main()
{
    int x;
    cout << "Enter a number: ";
    cin >> x;

    int y;
    cout << "Enter another number: ";
    cin >> y;

    int op;
    cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
    cin >> op;

    int result;
    switch (op)
    {
        // 使用早绑定来直接调用函数
        case 0: result = add(x, y); break;
        case 1: result = subtract(x, y); break;
        case 2: result = multiply(x, y); break;
    }

    cout << "The answer is: " << result << endl;

        return 0;
}
```

由于上面三个函数的调用都是直接使用函数名，采用早捆绑的方式。编译器会将每个函数调用替换为一个跳转指令，这个指令告诉 CPU 跳转到函数的地址来执行。

但是有时候，我们在程序运行前并不知道调用哪个函数，此时必须使用晚捆绑或者动态捆绑。晚绑定的一个例子就是使用函数指针，修改上面的例子：

```
int main()
{
    int x;
```

```
    cout << "Enter a number: ";
    cin >> x;

    int y;
    cout << "Enter another number: ";
    cin >> y;

    int op;
    cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
    cin >> op;

    // 定义一个函数指针
    int(*opFun)(int, int) = nullptr;
    switch (op)
    {
        // 使用早捆绑来直接调用函数
        case 0: opFun = add; break;
        case 1: opFun = subtract; break;
        case 2: opFun = multiply; break;
    }

    // 通过函数指针来调用，只能是晚捆绑
    cout << "The answer is: " << opFun(x, y) << endl;

        return 0;
}
```

使用函数指针来间接调用函数，编译器在编译阶段并不知道函数指针到底指向哪个函数，所以必须使用动态捆绑的方式。

动态绑定看起来更灵活，但是其是有代价的。静态捆绑时，CUP 可以直接跳转到函数地址。但是动态捆绑，CPU 必须先提取指针的地址，然后再跳转到指向的函数地址。这多了一个步骤！

虚函数表（Vtable）

C++使用了一种称为“虚表”的晚捆绑技术来实现虚函数。虚表是一个函数查询表，以动态捆绑的方式解析函数调用。每个具有一个或者多个虚函数的类都有一张虚表，这个表是在编译阶段建立的静态数组，其中包含了每个虚方法的函数指针，这些指针指向的是该类可见的派生最远的函数实现。其次，编译器会在基类对象都会添加一个隐含指针，这里我们称为*__vptr。这个指针当然能够被派生类所继承，这相当重要。当类的实例被创建时，这个指针指向该类所对应的虚表。这样，当使用某个对象调用虚方法时，通过该指针查找虚表，然后根据实际的对象类型执行正确版本的方法调用。看下面的简单例子：

```
class Base
{
public:
    virtual void function1() { }
    virtual void function2() { }
```

```
}

class D1: public Base
{
public:
    virtual void function1() override { }
}

class D2: public Base
{
public:
    virtual void function2() override { }
}
```

上面包含 3 个类，其中派生类 D1 与 D2 分别重写了基类的 function1()和 function2()虚方法。编译器会相应地创建 3 个不同的虚表，分别对应每个类。而且编译器也会自动地为基类添加一个函数指针，如下所示：

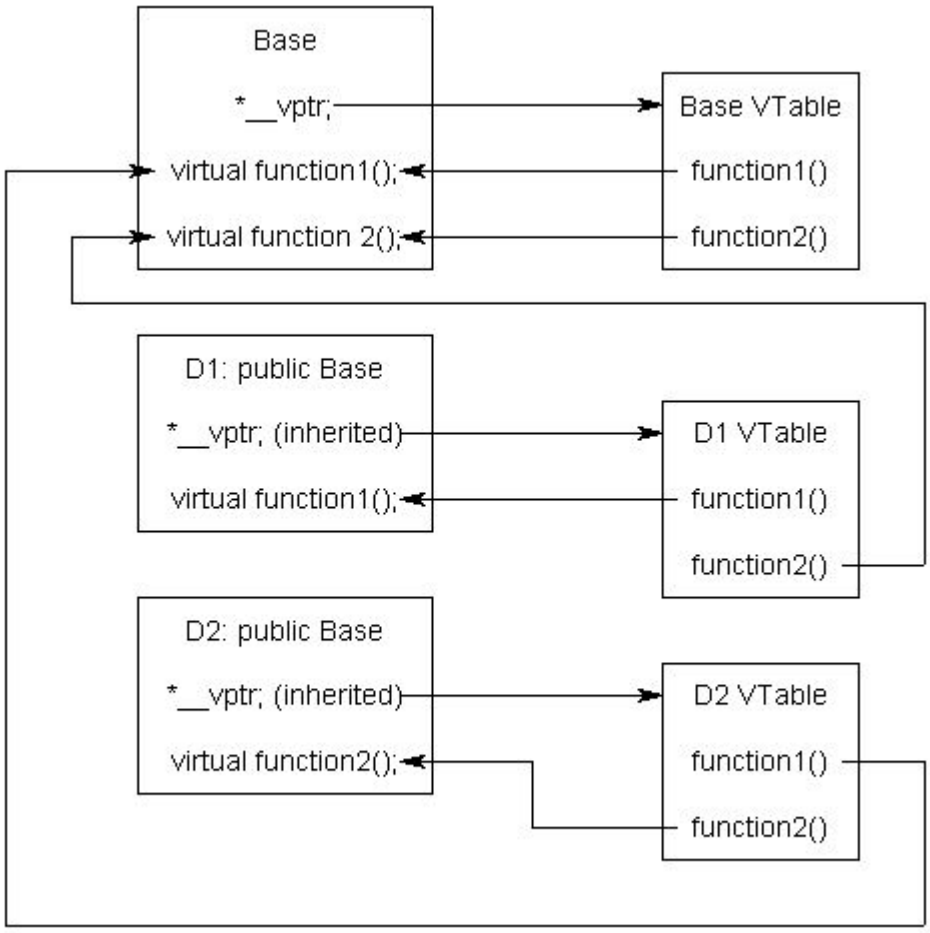
```
class Base
{
public:
    FunctionPointer *__vptr;
    virtual void function1() { }
    virtual void function2() { }
}

class D1: public Base
{
public:
    virtual void function1() override { }
}

class D2: public Base
{
public:
    virtual void function2() override { }
}
```

这样，每个类实例创建时，*__vptr 将指向该类所对应的虚表，比如基类的一个实例创建时，这个指镇就指向基类的虚表。

下面我们看看每个类的虚表是怎么建立的。因为仅有两个虚方法，所以每个虚表仅包含两个函数指针，分别对应 function1()和 function2()。但是每个函数指针实际指向的是那个类所可见的派生最远的函数实现：

- Base 的虚表：因为 Base 的实例仅可见自己的成员，所以它的虚表中的指针分别指向 Base::function1()和 Base::function2()；
- D1 的虚表：D1 的实例可见 Base 的成员与自身的成员，但是 D1 仅重写了 function1()，所以虚表中的指针分别指向 D1::function1()和 Base::function2()；

- D2 的虚表：与 D1 类似，分别指向 Base::function1()和 D2::function2()。
  下面是具体的示意图（来源：learncpp）：



所以，下面的代码就有了很好的解释：

```
int main()
{
    D1 d1;    // d1 中的*__vptr 指向类 D1 的虚表
    Base *dPtr = &d1;   //dPtr 对*__vptr 是可见的，但是实际上其指向的是 D1 的虚表；
    dPtr->function1();   // 此时 dPtr 通过虚表查找，调用的是 D1::function1()
}
```

使用虚表技术，虚函数得以正确实现！从而实现多态性！

纯虚函数与抽象基类

有时候，基类的某个虚方法并不需要实现，但是希望派生类能够提供重写的版本。这个时候，你需要定义纯虚函数。纯虚函数在类的定义中显示说明该方法不需要实现，其作用在于指明派生类必须要重写它。纯虚函数的定义很简单：方法声明后紧跟着=0。如果一个类中至少含有一个纯虚函数，那么这个类是抽象基类，因为这个类无法实例化。当继承一个抽象类时，必须重写所有纯虚函数，否则继承出来的类也是一个抽象类。下面演示例子：

```
class Animal
{
```

```
public:
    Animal(const string& name):
        m_name{name}
    {}

    const string& getName() const
    {
        return m_name;
    }

    virtual string speak() const = 0;  // 纯虚函数
    // 因为包含一个纯虚方法，所以是抽象基类

private:
    string m_name;
};

class Cat : public Animal
{
public:
    Cat(const string& name):
        Animal(name)
    {}

    // 重写了纯虚方法，所以 Cat 不是抽象类，可以实例化
    virtual string speak() const
    {
        return "Meow";
    }
};

// Dog 没有重写基类的纯虚方法，所以仍然无法实例化
class Dog : public Animal
{
public:
    Dog(const string& name):
        Animal(name)
    {}
};

int main()
{
    // Animal animal{"luly"}; // 无法编译，因为抽象基类无法实例化
    Cat cat{ "Sally" };      // 合法
```

```
    // Dog dog{ "Betsy" };      // 非法，抽象类无法实例化

    // 下面的代码可以运行，因为可以指向可以实例化的派生类对象
    Animal* aPtr = new Cat{ "Sally" };
    cin.ignore(10);
        return 0;
}
```

抽象类至少包含一个纯虚方法，抽象类提供了一种禁止其他代码直接实例化对象的方法，但是重写纯虚方法的派生类可以实例化。

接口类

接口是一个抽象的概念，使用者只关注功能而不要求了解实现。一个接口类可以看成一些纯虚方法的集合，这意味着接口类仅有定义功能，而没有具体的实现。C++ 其实没有单独的接口概念，而在 Java 和 C#等语言中接口是与类相区别的。但是 C++ 仍然可以使用接口类实现类似的效果。有时候，我们也称接口类为纯抽象类，因为这个类中全是虚方法。下面是一个纯抽象类的例子：

```
// 乐器纯抽象类
class Instrument
{
public:
    virtual void play() const = 0;
    virtual string what() const = 0;
    virtual void adjust(int) = 0;
};

class Wind: public Instrument
{
public:
    virtual void play() const override
    {
        cout << "Wind: paly" << endl;
    }

    virtual string what() const override
    {
        return "Wind";
    }

    virtual void adjust(int i) override {}
};

class Brass : public Instrument
{
public:
    virtual void play() const override
```

```
    {
        cout << "Brass: paly" << endl;
    }

    virtual string what() const override
    {
        return "Brass";
    }

    virtual void adjust(int i) override {}
};

void tune(Instrument& i)
{
    // ...
    i.play();
}

void f(Instrument& i)
{
    i.adjust(1);
}

int main()
{

    Wind wind;
    Brass brass;
    tune(wind);
    tune(brass);
    f(wind);
    f(brass);
        return 0;
}
```

可以看到 Instrument 是一个纯抽象类，其只提供方法的声明，具体却没有实现。但是它的两个派生类分别重写了这些纯虚方法，因此可以实例化。并且两个函数可以接收任意继承了 Instrument 的类实例对象。进一步说，这两个函数仅关注接收的对象是否提供了 Instrument 所要求的接口，但是不关注具体是怎么实现的。纯抽象类提供了更高级的抽象！这符合 OOP 的思想。

虚基类

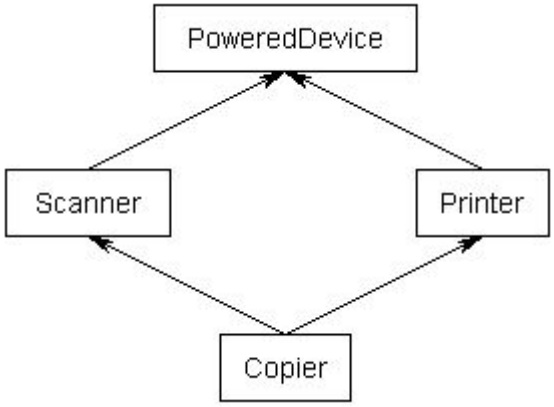
虚基类主要是用来解决菱形层次结构中的歧义基类问题。菱形层次结构是多重继承中的一个典例，还是例子说话：

```
class PoweredDevice
{
```

```
public:
    PoweredDevice(int power)
    {
        cout << "PoweredDevice: " << power << endl;
    }

    virtual void reportError() { cout << "Error" << endl; }
};

class Scanner : public PoweredDevice
{
public:
    Scanner(int scanner, int power) :
        PoweredDevice(power)
    {
        cout << "Scanner: " << scanner << endl;
    }
};

class Printer : public PoweredDevice
{
public:
    Printer(int printer, int power) :
        PoweredDevice(power)
    {
        cout << "Printer: " << printer << endl;
    }
};

class Copier : public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power):
        Scanner(scanner, power), Printer(printer, power)
    {}
};
int main()
{
    Copier copier(1, 2, 3);
    // output:
    // PoweredDevice: 3
    // Scanner : 1
    // PoweredDevice : 3
    // Printer : 2
```

```
    // 可以看到 PoweredDevice 被继承了两次

    // 无法编译，有歧义，因为继承了两个版本的 PoweredDevice
    copier.reportError();
        return 0;
}
```

上面的继承关系有点复杂，但是画出继承图谱（来源：learncpp）就很清晰了：



Scanner 和 Printer 分别继承了 PoweredDevice 类，然后 Copier 又同时继承了 Scanner 和 Printer 类，我们实际希望 Copier 仅继承一次 PoweredDevice 类，但是实际上 Copier 包含了两个版本的 PoweredDevice。所以可以看到，次 PoweredDevice 被构造了两次。而且更严重的是，PoweredDevice 中没有被重写的方法是无法调用的，因为编译器会给出一个有歧义的错误！解决这个错误的方法很多，比如你可以在 Copier 类中明确声明继承的版本：using Scanner::PoweredDevice::reportError;。但是这本质上没有解决多版本的继承问题。

此时，你可以用虚基类，使用虚基类，只需要在继承列表中加上 virtual 关键字：

```
class PoweredDevice
{
public:
    PoweredDevice(int power)
    {
        cout << "PoweredDevice: " << power << endl;
    }

    virtual void reportError() { cout << "Error" << endl; }
};

class Scanner : virtual public PoweredDevice
{
public:
    Scanner(int scanner, int power) :
        PoweredDevice(power)
    {
```

```
        cout << "Scanner: " << scanner << endl;
    }
};

class Printer : virtual public PoweredDevice
{
public:
    Printer(int printer, int power) :
        PoweredDevice(power)
    {
        cout << "Printer: " << printer << endl;
    }
};

class Copier : public Scanner, public Printer
{
public:
    // Note: 虚基类是由派生最远的类负责创建，所以，
    //       构造函数初始化列表中需要增加虚基类的构造函数调用
    Copier(int scanner, int printer, int power):
        Scanner(scanner, power), Printer(printer, power),
        PoweredDevice(power)
    {}

};
int main()
{
    Copier copier(1, 2, 3);

    // 合法
    copier.reportError();

    // output:
    // PoweredDevice: 3
    // Scanner : 1
    // Printer : 2
    // 可以看到 PoweredDevice 继承了一次

    return 0;
}
```

利用虚基类，可以解决上面多重继承中歧义基类问题，基类仅被继承一次。但是要注意的是此时的虚基类由派生最远的类负责创建（可以看成该类的直接基类），因为 PoweredDevice 并没有无参构造函数，所以在 Copier 构造函数初始化列表中必须加上 PoweredDevice 的有参构造函数调用！

说点题外话，尽管虚基类可以解决多重继承中的菱形层次结构，但是看起来还是很抽象与复杂。实际上，多重继承本来就是一个很有争议的话题，因为使用多重继承会使得继承体系变得复杂，而且产生一系列问题，像 Java 和 C#这类语言，是不允许多重继承的，但是其单独提供了接口，类可以继承多个接口，这也相当于多重继承了。而且好处是接口的继承相当于组合，这也是比较推崇的！

对象切片

前面讲过，实现虚函数及多态性必须要用传地址的方式（引用或者指针）。一般，地址具有相同的长度，这意味着派生类对象的地址与基类对象的地址也是相同，尽管派生类对象所占的内存一般要高过基类对象。所以，传地址的方式不会导致类型信息损失，进而可以实现多态性。看下面的例子：

```
class Base
{
public:
    Base(int value):
        m_value{value}
    {}

    virtual string getName() const { return "Base"; }
    int getValue() const { return m_value; }
protected:
    int m_value;
};

class Derived: public Base
{
public:
    Derived(int value):
        Base(value)
    {}

    virtual string getName() const override { return "Derived"; }
}

int main()
{
    Derived derived{ 5 };
    cout << "derived is a " << derived.getName() << " with value " <<
derived.getValue() << endl;
    // output: derived is a Derived with value 5
    Base& ref = derived;
    cout << "ref is a " << ref.getName() << " with value " << ref.getValue() << endl;
    // output: ref is a Derived with value 5
    Base* ptr = &derived;
```

```
    cout << "ptr is a " << ptr->getName() << " with value" << ptr->getValue() <<
endl;
    // output: ptr is a Derived with value 5
    Base base = derived;
    cout << "base is a " << base.getName() << " with value " << base.getValue() <<
endl;
    // output: base is a Base with value 5
    return 0;
}
```

可以看到使用引用或者指针的方式，多态性都能够实现，但是传值的方式就存在问题。当我们将一个派生类对象直接赋值给基类对象时，仅仅基类的部分被复制，派生类的那部分信息将丢失。我们称这种现象为“对象切片”：对象丢失了自己原有的部分信息。使用对象本身并没有问题，但是处理不当，会造成很多问题，看下面的例子：

```
int main()
{
    Derived d1{5};
    Derived d2{2};
    Base& b = d2;
    b = d1;   // 有隐患
    return 0;
}
```

上面的例子很简单，但是会有问题：首先 d2 引用给 b 时，b 将指向 d2，这没有问题。但是将 d1 的值直接赋值给 b 时，会发生对象切片，只有 d1 的基类部分复制给 b。此时，问题来了，你会发现现在 d2 拥有 d1 的基类部分与 d2 的派生部分，这显得很混乱！所以，尽可能地别使用对象切片，否则你会麻烦不断！

动态转型

前面的例子，我们都是将派生类对象复制给基类对象，不管是通过传地址的方式还是对象切片方式。这些都是向上转型——在类层次中向上移动。我们不禁会想，肯定会存在可以向下移动的向下转型。一般来说，派生类包含基类信息，所以向上转型是容易的。但是，反过来可能会失败！因为无法保证基类对象实际上存储的是派生类对象。看下面的例子：

```
void process(Base* ptr)
{
    Derived* derived = static_cast<Derived*>(ptr);
    // 后序处理
    // ...
}
```

process 函数接收一个基类指针，但是在内部使用 static_cast 向下转型为派生类指针，然后进行后序处理。如果送入 process 函数的指针实际上就是指向派生类对象，那么上面的代码是没有问题的。但是，如果仅仅传入就是指向基类对象的指针，或者指向其他派生类的指针，那么函数内部的转型将存在问题：由于 static_cast 在运行时是不检查对象实际类型的，这将导致不可控行为！

为了解决这样的隐患，C++引入了运行时的动态类型转化操作符 dynamic_cast。

dynamic_cast 在运行时检测底层对象的类型信息。如果类型转换没有意义，那么它将返回

一个空指针（对于指针类型）或者抛出一个 std::bad_cast 异常（对于引用类型）。所以，可以修改上面的代码如下：

```
void process(Base* ptr)
{
    Derived* derived = dynamic_cast<Derived*>(ptr);
    if (derived == nullptr)
    {
        // 后序处理
        // ...
    }
}
```

尽管如此，向下转型还是不推荐的，除非必要！

Reference

[1] cpp leraning online（本文按照该教程书写，作者人很 nice，可以直接留言）.

[2] Marc Gregoire. Professional C++, Third Edition, 2016.

[3] cppreference

[4] Bruce Eckel, Chuck Allison. Thinking in C++, Second Edition, 2011.