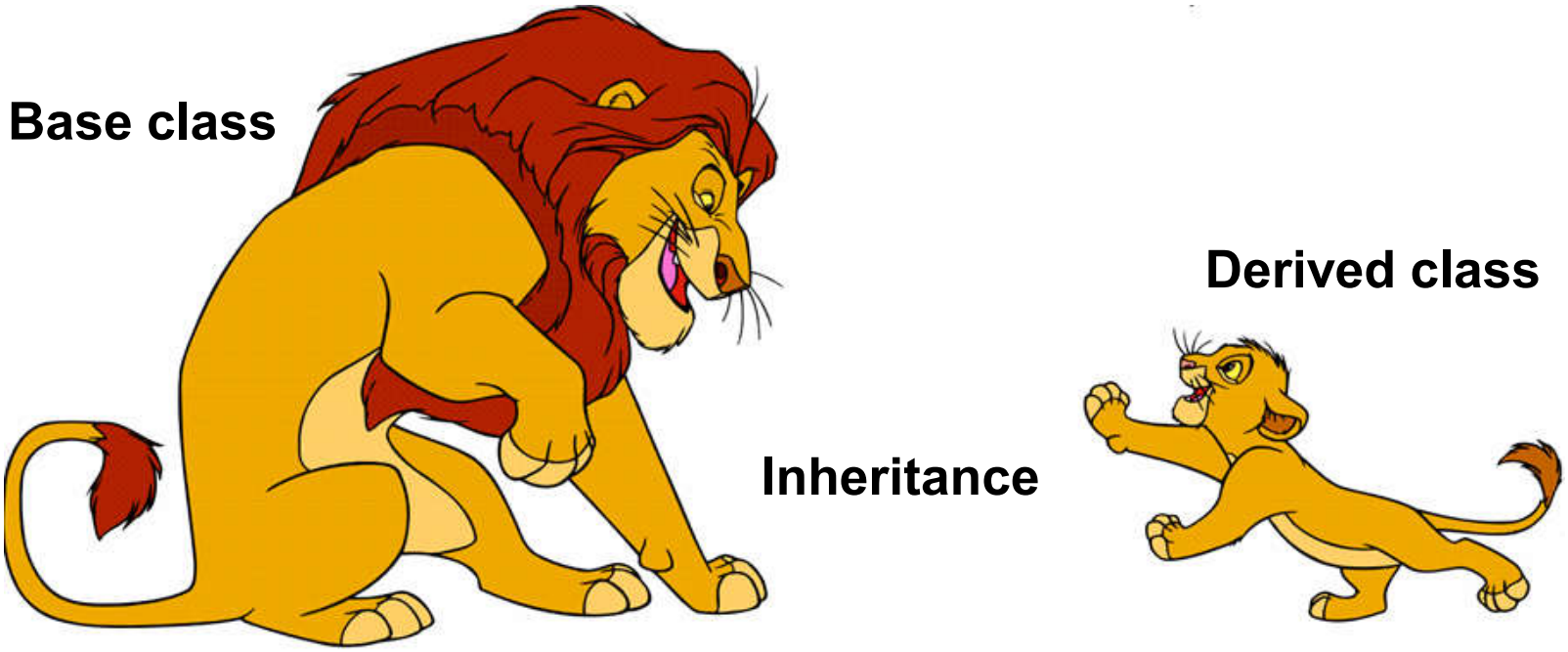
Inheritance

ThanhNT

Introduction to inheritance

Inheritance in real life

We meet inheritance everywhere in real life. For example, the child inherited all the properties of his farther



Introduction to inheritance

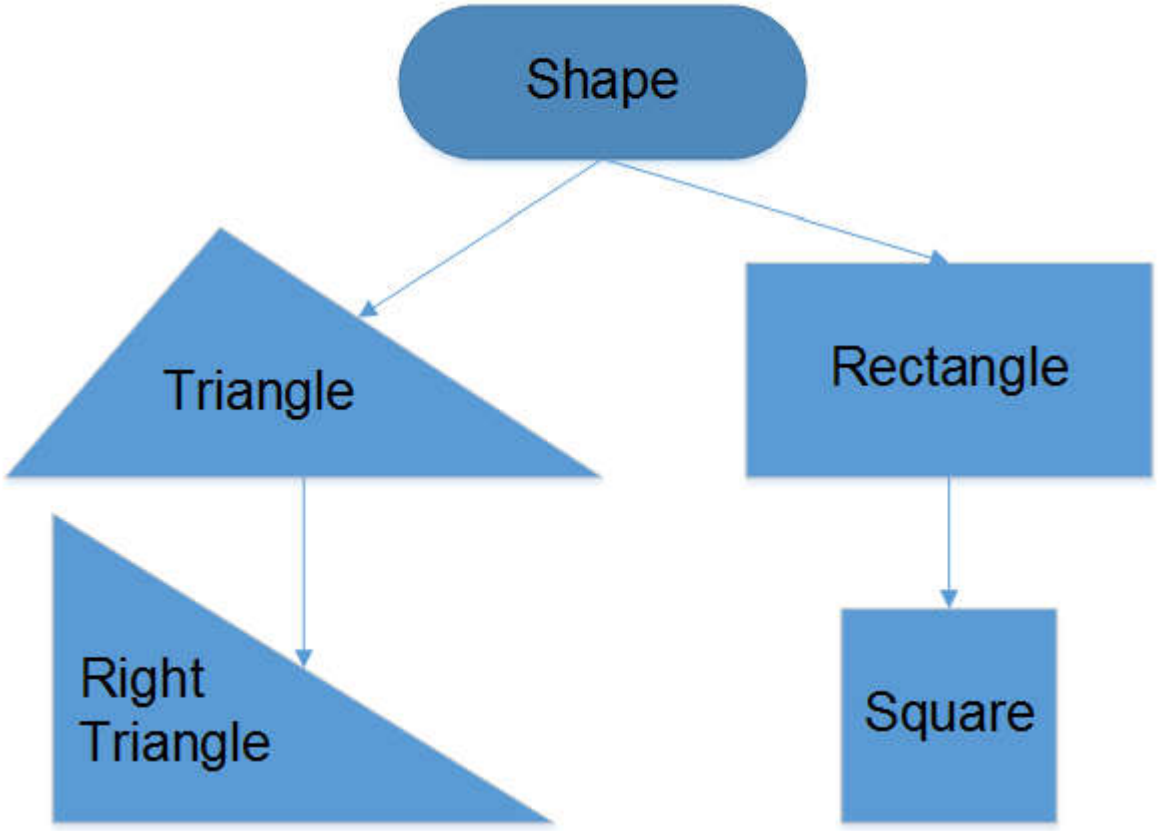
Inheritance in C++

Inheritance is capability of one class to acquire the attributes and behaviors from other classes.

Base class

Derived & Base class

Derived class



Introduction to inheritance

Why the need for inheritance in C++?

The existing code often does not do **EXACTLY** what we want, how do we reuse the code?

- Change the existing code
  - Cannot use the existing code for it's original purpose.
- Copy then change by search/replace
  - Omitted or misplaced
  - Inadvertently replace some thing you didn't mean.
  - Duplicate
- Inheritance
  - Reuse directly the existing code
  - Add new features
  - Redefine existing features
  - Hide existing features

Basic inheritance in C++

Simple Base class

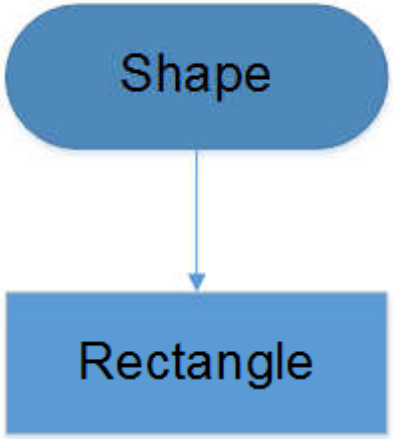
```cpp
#include <iostream>
#include <string>
class Shape
{
public:
    std::string m_strColor;
    float m_area;
    float m_perimeter;
    std::string GetColor(){return m_strColor;}
    float GetArea(){return m_area;}
    float GetPerimeter(){return m_perimeter;}
    Shape(std::string strColor = "", float area = 0, float perimeter = 0)
        :m_strColor(strColor), m_area(area), m_perimeter(perimeter)
    {
        std::cout << "Shape -- Base" << std::endl;
    }
};
```

This base class hold information about shape(color, area, perimeter) that are common to all shapes

Basic inheritance in C++

Simple Derived class

```
class Rectangle : public Shape
{
public:
    float m_width;
    float m_length;
    Rectangle(float width = 0, float length = 0)
        :m_width(width), m_length(length)
    {
        std::cout << "Rectangle - Derived" << std::endl;
    }
};
```



Rectangle inherits from **Shape**, It automatically receives the function and variable from Shape. Thus Rectangle will have 5 member variables(**m_width**, **m_length** from Rectangle, and **m_strColor**, **m_area**, and **m_perimeter** from Shape)

Basic inheritance in C++

Instantiate a derived class

```
int main()
{
    std::string rectColor;
    /* use derived constructor */
    Rectangle blueRect;
    blueRect.m_strColor = "blue";
    rectColor = blueRect.GetColor();
    std::cout << "rectangle color is " + rectColor << std::endl;
    /* return code */
    return 0;
}
```

Which prints the result

```
Shape -- Base
Rectangle -- Derived
rectangle color is blue
[Finished in 0.4s]
```

The base class is constructed first, then derived class is constructed

Constructor and initialization of derived classes

Constructor of derived classes

With non-derived class, constructors only have worry about their own members.

```
int main()
{
    /* Base class constructor */
    Shape cShape("red", 0, 0);
    /* return code */
    return 0;
}
```

- Memory for cShape is set aside
- The appropriate Base constructor is called
- The initialization list initializes variables
- The body of the constructor executes
- Control is returned to the caller

Constructor and initialization of derived classes

Constructor of derived classes

With derived classes, things are slightly more complex

```
int main()
{
    /* derived class constructor */
    Rectangle blueRect(0, 0);
    /* return code */
    return 0;
}
```

- Memory for blueRect is set aside (enough for both the Base and Derived portions).
- The appropriate Derived constructor is called
- **The Base object is constructed first using the appropriate Base constructor**
- The initialization list initializes variables
- The body of the constructor executes
- Control is returned to the caller

Constructor and initialization of derived classes

Initializing base class members

With current Rectangle derived class as written is that there is no way to initialize base class members when we create a derived class.

```
class Rectangle : public Shape
{
public:
    float m_width;
    float m_length;
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :m_width(width), m_length(length)
    {
        m_strColor = strColor;
        std::cout << "Rectangle - Derived" << std::endl;
    }
};
```

C++ allows inherited variables can still have their value changed in the body of constructor using an assignment.

Constructor and initialization of derived classes

Initializing base class members

But the problem still persist if **m_strColor** is const or reference variable that requires initializing in the initialization list of the constructor.

```
class Rectangle : public Shape
{
public:
    float m_width;
    float m_length;
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :Shape(strColor), m_width(width), m_length(length)
    {
        std::cout << "Rectangle - Derived" << std::endl;
    }
};
```

C++ gives us the ability to explicitly choose which Base class constructor will be called.

Constructor and initialization of derived classes

Initializing base class members

Now execute this code

```
int main()
{
    std::string rectColor;
    /* derived class constructor */
    Rectangle blueRect("blue", 0, 0);
    std::cout << "rectangle color is " + rectColor << std::endl;
    rectColor = blueRect.GetColor();
    /* return code */
    return 0;
}
```

Which prints the result

```
Shape -- Base
Rectangle -- Derived
rectangle color is blue
[Finished in 0.4s]
```

Inheritance and access specifier

In Shape class example, we mark all members(m_strColor, m_area, m_perimeter) are public. Now let's do a little change.

```
#include <iostream>
#include <string>
class Shape
{
private:
    std::string m_strColor;
protected:
    float m_area;
public:
    float m_perimeter;
    std::string GetColor(){return m_strColor;}
    float GetArea(){return m_area;}
    float GetPerimeter(){return m_perimeter;}
    Shape(std::string strColor = "", float area = 0, float perimeter = 0)
        :m_strColor(strColor), m_area(area), m_perimeter(perimeter)
    {
        std::cout << "Shape - Base" << std::endl;
    }
};
```

Inheritance and access specifier

Now instantiate a base instance with the change

```
int main()
{
    Shape blueShape;
    /* not allow, cannot access private members from outside of class */
    blueShape.m_strColor = "blue";
    /* not allow, cannot access protected members from outside of class */
    blueShape.m_area = 10;
    /* allow, anybody can access public members */
    blueShape.m_perimeter = 8;
    /* return code */
    return 0;
}
```

From outside of class, user can only access public members.

How about derived class?

Inheritance and access specifier

Public inheritance

```
class Rectangle : public Shape
{
public:
    float m_width;
    float m_length;
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :m_width(width), m_length(length)
    {
        /* not allow, derived class cannot access private members in base class */
        m_strColor = "blue";
        /* allow, derived class can access to protected members in base class */
        m_area = 10;
        /* allow, anybody can access to public members */
        m_perimeter = 8;
        std::cout << "Rectangle - Derived" << std::endl;
    }
};
```

Derived class can access to **public** and **protected** members in base class.

Inheritance and access specifier

Public inheritance

Instantiate an instance of derived class

```
int main()
{
    Shape blueShape;
    /* not allow, cannot access private members from outside of class */
    blueShape.m_strColor = "blue";
    /* not allow, cannot access protected members from outside of class */
    blueShape.m_area = 10;
    /* allow, anybody can access public members */
    blueShape.m_perimeter = 8;
    /* return code */
    return 0;
}
```

To summarize in table form:

Public inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Public	Yes	Yes
Private	Private	No	No
Protected	Protected	Yes	No

Inheritance and access specifier

Private inheritance

```
class Rectangle : private Shape
{
public:
    float m_width;
    float m_length;
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :m_width(width), m_length(length)
    {
        /* not allow, derived class cannot access private members in base class */
        m_strColor = "blue";
        /* allow, derived class can access to protected members in base class */
        m_area = 10;
        /* allow, anybody can access to public members */
        m_perimeter = 8;
        std::cout << "Rectangle - Derived" << std::endl;
    }
};
```

Derived class can access to **public** and **protected** members in base class.

Inheritance and access specifier

Private inheritance

Instantiate an instance of derived class

```
int main()
{
    Rectangle priRect;
    /* not allow, cannot access private members from outside of class */
    priRect.m_strColor = "blue";
    /* not allow, protected member is now a private member when access through derived class */
    priRect.m_area = 10;
    /* not allow, public member is now a private member when access through derived class */
    priRect.m_perimeter = 8;
    /* return code */
    return 0;
}
```

To summarize in table form:

Private inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Private	Yes	No
Private	Private	No	No
Protected	Private	Yes	No

Inheritance and access specifier

Protected inheritance

Protected inheritance			
Base access specifier	**Derived access specifier**	**Derived class access?**	**Public access?**
Public	Protected	Yes	No
Private	Private	No	No
Protected	Protected	Yes	No

Adding, changing and hiding members in a derived class

Adding new functionality

To add new functionality to a derived class, simply declare that functionality in the derived class

```
class Rectangle : public Shape
{
private:
    float m_width;
    float m_length;
public:
    void Identify(){std::cout << "I am a derived" << std::endl;}
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :Shape(strColor), m_width(width), m_length(length)
    {
    }
};
```

Adding, changing and hiding members in a derived class

Adding new functionality

Now new public will be able to call Identify() to identify the class.

```
int main()
{
    Rectangle bleRect;
    bleRect.Identify();
    /* return code */
    return 0;
}
```

This produces the result

```
Shape - Base
I am a derived
[Finished in 0.4s]
```

Adding, changing and hiding members in a derived class

Redefining functionality

Shape class defines GetArea() function, and we can redefine it in the derived class.

```
class Rectangle : public Shape
{
private:
    float m_width;
    float m_length;
public:
    void Identify(){std::cout << "I am a derived" << std::endl;}
    float GetArea(){return m_width*m_length;};
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :Shape(strColor), m_width(width), m_length(length)
    {
    }
};
```

Adding, changing and hiding members in a derived class

Redefining functionality

Now use new functionality to calculate area of the rectangle

```
int main()
{
    Rectangle blueRect("blue", 5, 8);
    blueRect.Identify();
    std::cout << blueRect.GetArea() << std::endl;
    /* return code */
    return 0;
}
```

This produces the result

```
Shape -- Base
I am a derived
40
[Finished in 0.4s]
```

Adding, changing and hiding members in a derived class

Adding to existing functionality

C++ allows our **Derived** function to call the **Base** function of the same name.

```
class Rectangle : public Shape
{
private:
    float m_width;
    float m_length;
public:
    void Identify(){std::cout << "I am a derived" << std::endl;}
    float GetArea(){
        std::cout << "Base area = " << Shape::GetArea() << std::endl;
        return m_width*m_length;
    };
    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :Shape(strColor), m_width(width), m_length(length)
    {
    }
};
```

Adding, changing and hiding members in a derived class

Adding to existing functionality

Now consider the following example

```
int main()
{
    float blueRectArea;
    Rectangle blueRect("blue", 5, 8);
    blueRectArea = blueRect.GetArea();
    std::cout << "Rectangle area = " << blueRectArea << std::endl;
    /* return code */
    return 0;
}
```

This produces the result

```
Shape - Base
Base area = 0
Rectangle area = 40
[Finished in 0.4s]
```

Adding, changing and hiding members in a derived class

Hiding functionality

In C++, it is possible to hide existing functionality from a class.

```
class Rectangle : public Shape
{
private:
    float m_width;
    float m_length;
public:
    void Identify(){std::cout << "I am a derived" << std::endl;}
    float GetArea(){
        std::cout << "Base area = " << Shape::GetArea() << std::endl;
        return m_width*m_length;
    };

    Rectangle(std::string strColor = "", float width = 0, float length = 0)
        :Shape(strColor), m_width(width), m_length(length)
    {
    }
private:
    Shape::GetPerimeter;
};
```

Adding, changing and hiding members in a derived class

Hiding functionality

Now try to call GetPerimeter() from public.

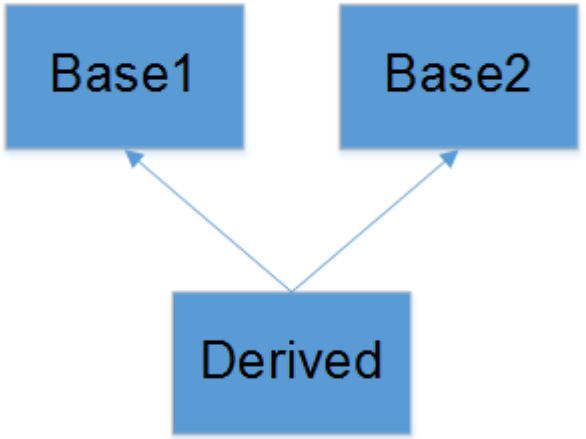
```
int main()
{
    float blueRectPerimeter;
    Rectangle blueRect("blue", 5, 8);
    /* will not work because GetPerimeter() has been redefined as private */
    blueRectPerimeter = blueRect.GetPerimeter();
    /* return code */
    return 0;
}
```

Of course, we can hide member variables by the same way.

Multiple inheritance

C++ provides the ability to do multiple inheritance mean a **Derived** class can inherit members from more than one **Base** class

```
class Base1
{
public:
    int m_base1Value;
    int GetValue(){return m_base1Value;}
    Base1(int base1Value = 0):m_base1Value(base1Value){/* body */}
};
class Base2
{
public:
    int m_base2Value;
    int GetValue(){return m_base2Value;}
    Base2(int base2Value = 0):m_base2Value(base2Value){/* body */}
};
class Derived : public Base1, public Base2
{
public:
    int m_derivedValue;
    Derived(int derivedValue = 0):m_derivedValue(derivedValue){/* body */}
};
```



Multiple inheritance

Now instantiate a derived instance

```
int main()
{
    Derived testDerived;
    testDerived.m_base1Value = 1;
    std::cout << "Base 1 value = " << testDerived.m_base1Value << std::endl;
    testDerived.m_base2Value = 2;
    std::cout << "Base 2 value = " << testDerived.m_base2Value << std::endl;
    return 0;
}
```

Which prints the result

```
Base 1 value = 1
Base 2 value = 2
[Finished in 0.4s]
```

Multiple inheritance

Problems with multiple inheritance

Notice that both Base1 and Base2 classes have the same function GetValue() so which function will be called in below example?

```
int main()
{
    Derived testDerived;
    std::cout << "Member value = " << testDerived.GetValue() << std::endl;
    return 0;
}
```

The compiler will report error

```
C:\Users\public\Desktop\shape.cpp: In function 'int main()':
C:\Users\public\Desktop\shape.cpp:26:51: error: request for member 'GetValue' is ambiguous
     std::cout << "Member value = " << testDerived.GetValue << std::endl;
                                                   ^
C:\Users\public\Desktop\shape.cpp:14:9: note: candidates are: int Base2::GetValue()
     int GetValue(){return m_base2Value;}
         ^
C:\Users\public\Desktop\shape.cpp:7:9: note:                 int Base1::GetValue()
     int GetValue(){return m_base1Value;}
         ^
```

Multiple inheritance

Problems with multiple inheritance

However, we can explicitly specify which base class we meant to call to resolve above problem.

```
int main()
{
    Derived testDerived;
    std::cout << "Member value = " << testDerived.Base1::GetValue() << std::endl;
    return 0;
}
```

Which will print the result

```
Member value = 0
[Finished in 0.5s]
```

Virtual base classes

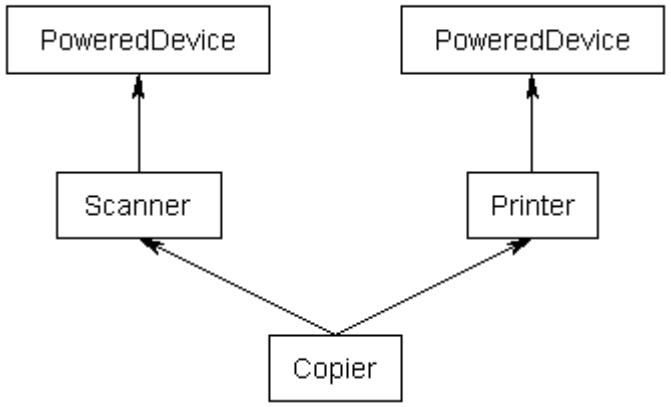
Virtual base classes example

```
class PoweredDevice
{
public:
    PoweredDevice(int nPower){cout << "PoweredDevice: " << nPower << endl;}
};
class Scanner: public PoweredDevice
{
public:
    Scanner(int nScanner, int nPower): PoweredDevice(nPower)
    {cout << "Scanner: " << nScanner << endl;}
};
class Printer: public PoweredDevice
{
public:
    Printer(int nPrinter, int nPower): PoweredDevice(nPower)
    {cout << "Printer: " << nPrinter << endl;}
};
class Copier: public Scanner, public Printer
{
public:
    Copier(int nScanner, int nPrinter, int nPower)
    : Scanner(nScanner, nPower), Printer(nPrinter, nPower){}
};
```

Virtual base classes

Now if you create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner

```
int main()
{
    Copier cCopier(1, 2, 3);
    return 0;
}
```



Which produces the result

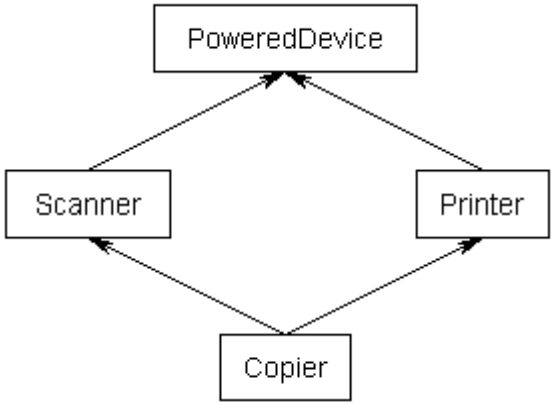
```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
[Finished in 0.4s]
```

PoweredDevice got constructed **twice**

Virtual base classes

Sometime, we may want only one **PoweredDevice** to be shared by both Scanner and Printer and we create a virtual base class.

```
class PoweredDevice
{
};
class Scanner: virtual public PoweredDevice
{
};
class Printer: virtual public PoweredDevice
{
};
class Copier: public Scanner, public Printer
{
};
```



But who (Scanner or Printer) is responsibility for creating PowerdDevice base class? The answer is Copier.

Virtual base classes

```
class Copier: public Scanner, public Printer
{
public:
    Copier(int nScanner, int nPrinter, int nPower)
        : Scanner(nScanner, nPower), Printer(nPrinter, nPower), PoweredDevice(nPower)
    {
    }
};
int main()
{
    Copier cCopier(1, 2, 3);
    return 0;
}
```

Which prints the result

```
PoweredDevice: 3
Scanner: 1
Printer: 2
[Finished in 0.4s]
```

Now PoweredDevice only gets constructed **once**

Virtual base classes

- **Virtual** base classes are created before **non-virtual** base classes.
- Note that the **Scanner** and **Printer** constructors still have calls to the **PoweredDevice** constructor. If we are creating an instance of **Copier**, these constructor calls are simply **ignored**.
- If we were to create an instance of **Scanner** or **Printer**, the **virtual** keyword is ignored, and **normal inheritance rules** apply.
- If a class inherits **one or more classes** that have virtual parents, the most **derived** class is **responsible** for constructing the virtual base class.

Thanks for your attention!