一、虚函数

定义：在类的定义中，**前面带有`virtual`关键字的成员函数**就是虚函数。如下：

```
class Student{
    public:
        virtual int get();
};
int Student::get(){}
```

`virtual`**只在声明函数时使用**，定义时则不用加

二、多态的表现形式1.0

- 派生类的指针可以赋给基类指针。
- 通过基类指针调用基类和派生类中的同名虚函数时
    1. 若该指针**指向一个基类的对象**，那么被调用是**基类的虚函数。**
    2. 若该指针**指向一个派生类的对象**，那么被调用的是**派生类的虚函数**

这种机制就叫做"**多态**"

```
class CBase{
  public:
    virtual void SomeVirtualFunction(){}
};
class CDerived:public CBase{
    public:
    virtual void SomeVirtualFunction(){}
};
int main()
{
    CDerived ODerived;
    CBase * p = &ODerived;
    p -> SomeVirtualFunction(); //调用那个函数取决与p指向的那种类型的对象
    return 0;
}
```

三、多态的表现形式2.0

- 派生类的对象可以赋给基类使用。
- 通过基类引用调用基类和派生类中的同名虚函数时：
    1. 若这个引用**引用的是一个基类的对象**，那么被调用的就是**基类的虚函数。**
    2. 若这个引用**引用的是一个派生类的对象**，那么被调用的就是**派生类的虚函数。**

这种机制也是**多态**

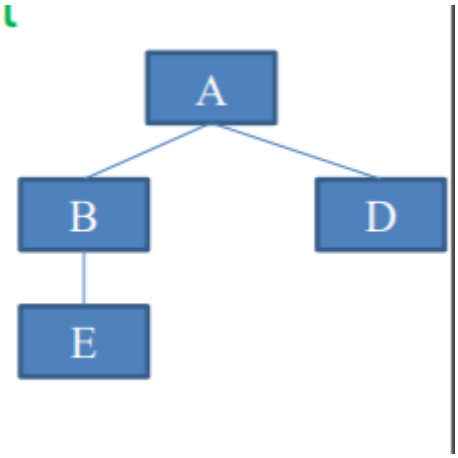
```
class CBase{
  public:
    virtual void SomeVirtualFunction(){}
};
```

```
class CDerived:public CBase{
    public:
    virtual void SomeVirtualFunction(){}
};
int main()
{
    CDerived ODerived;
    CBase & p = ODerived;
    p.SomeVirtualFunction(); //调用那个函数取决与p是那种类型的引用
    return 0;
}
```

多态的举例：

```
#include<bits/std++.h>
using namespace std;
class A{
    public:
        virtual void Print(){
            cout << "A.Print" << endl;
        }
};
class B : public A{
    public:
        virtual void Print(){
            cout << "B.Print" << endl;
    }
};
class D : public A{
    public:
        virtual void Print(){
            cout << "D.Print" << endl;
        }
};
class E : public B{
    public:
        virtual void Print(){
            cout << "E.Print" << endl;
        }
};
int main()
{
    A a;B b;E e;D d;
    A * pa = &a;
    B * pb = &b;
    D * pd = &d;
    E * pe = &e;
    pa->Print();
    pa = pb;
    pa-> Print();
    pa = pd;
    pa-> Print();
    pa = pe;
    pa-> Print();
    return 0;
}
//输出结果：
```

```
//A.Print
//B.Print
//D.Print
//E.Print
```



四、例题

几何形体处理程序: 输入若干个几何形体的参数， 要求按面积排序输出。输出时要指明形状。 Input: 第一行是几何形体数目n（不超过100).下面有n行，每行以一个字母c开头. 若 c 是 'R'，则代表一个矩形，本行后面跟着两个整数，分别是矩形的宽和高； 若 c 是 'C'，则代表一个圆，本行后面跟着一个整数代表其半径 若 c 是 'T'，则代表一个三角形，本行后面跟着三个整数，代表三条边的长度

Output: 按面积从小到大依次输出每个几何形体的种类及面积。每行一个几何形体。

```
Sample Input:
3
R 3 5
C 9
T 3 4 5


Sample Output
Triangle:6
Rectangle:15
Circle:254.34
```

AC代码：

```
#include<iostream>
#include<cstring>
#include<cstdio>
#include<cmath>
using namespace std;
class shape{
    public:
        virtual double area() = 0;
        virtual void print() = 0;
};
shape * p[50];
class rectangle : public shape{
    public:
        int w,h;
```

```
        virtual double area();
        virtual void print();
};
double rectangle :: area(){
    return w * h;
}
void rectangle :: print(){
    cout << "Rectangle:" << area() << endl;
}
class circle : public shape{
    public:
        int r;
        virtual double area();
        virtual void print();
};
double circle :: area(){
    return r * r * 3.14;
}
void circle :: print(){
    cout << "Circle:" << area() << endl;
}
class triangle : public shape{
    public:
        int a,b,c;
        virtual double area();
        virtual void print();
};
double triangle :: area(){
    double p = (a + b + c)/2.0;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}
void triangle :: print(){
    cout << "Triangle:" << area() << endl;
}
int main()
{
    int n;
    rectangle * pr;
    circle * pc;
    triangle * pt;
    cin >> n;
    for(int i = 1;i <= n;i++){
        char c;
        cin >> c;
        switch(c){
            case'R':
                pr = new rectangle();
                cin >> pr->w >> pr->h;
                p[i] = pr;
                break;
            case'C':
                pc = new circle();
                cin >> pc->r;
                p[i] = pc;
                break;
            case'T':
                pt = new triangle();
                cin >> pt->a >> pt->b >> pt->c;
```

```
                p[i] = pt;
                break;
            }
        }
        for(int i = 1;i <= n;i++){
            p[i]->print();
        }
        for(int i = 1;i < n;i++){
            if(p[i]->area() > p[i + 1]->area()) cout << 1 ;
            else cout << 0 << ' ';
        }
        return 0;
}
```

注意：用基类数组存放指向各种派生类对象的指针，然后遍历数组，就能对各个派生类对象进行各种操作，这是一种很常见的做法

五、其他例子及其说明

```
#include<bits/stdc++.h>
using namespace std;
class Base {
    public:
        void fun1() { fun2(); }//{this->fun2()} this是基类指针，fun2是虚函数，所以是一个多态
        virtual void fun2() { cout << "Base::fun2()" << endl; }
};
class Derived:public Base {
    public:
        virtual void fun2() { cout << "Derived:fun2()" << endl; }
};
int main()
{
    Derived d;
    Base * pBase = & d;
    pBase->fun1();
    return 0;
}
```

注意：在构造函数和析构函数中调用虚函数，不是多态。编译时即可确定，调用的函数是自己的类或基类中定义的函数，不会等到运行时才决定调用自己的还是派生类的函数。

```
#include<bits/stdc++.h>
using namespace std;
class myclass{
    public:
        virtual void hello(){
            cout << "hello from myclass" << endl;
        }
        virtual void bye(){
            cout << "bye from myclass" << endl;
        }
};
class son : public myclass{
    public:
```

```
        void hello(){
            cout << "hello from son" << endl;
        }
        son(){
            hello();
        }
        ~son(){
            bye();
        }
};
class grandson : public myclass{
    public:
        void hello(){
            cout << "hello from grandson" << endl;
        }
        void bye() { cout << "bye from grandson"<<endl;}
        grandson(){cout<<"constructing grandson"<<endl;}
        ~grandson(){cout<<"destructing grandson"<<endl;};
};
int main()
{
    grandson gson;
    myclass * pson;
    pson = &gson;
    pson->hello();//多态
    return 0;
}
```

注意：派生类中和基类中虚函数同名同参数表的函数，不加virtual也自动成为虚函数

```
class Base {
    private:
        virtual void fun2() { cout << "Base::fun2()" << endl; }
};
class Derived:public Base {
    public:
        virtual void fun2() { cout << "Derived:fun2()" << endl; }
};
    Derived d;
    Base * pBase = & d;
    pBase -> fun2();//error
```

编译出错是因为 `fun2()` 是 `Base` 的私有成员。即使运行到此时实际上调用的应该是 `Derived` 的公有成员 `fun2()` 也不行，因为语法检查是不考虑运行结果的。

如果 将 `Base` 中的 `private` 换成 `public`,即使 `Derived` 中的 `fun2()` 是 `private` 的，编译依然能通过，也能正确调用 `Derived::fun2()`。

可以理解为：**对于派生类的私有成员我们可以通过多态访问，但是对于基类的私有成员则不可以**