C++多态的介绍

多态的应用场景

基类和派生类中存在同名函数时，多态则可以实现当使用基类的指针或引用来定义派生类时调用类中的同名函数时，可以实现派生类各自调用各自的函数。
代码实例，假设要设计一个 Animal 的基类，派生类有 Cat 和 Dog，它们有共同的属性 makeSound,在基类和派生类中，makeSound 的实现是不同的，代码如下：
animal.h 代码如下：

```
7   class Animal{
8   public:
9       Animal(){
10          cout << "Animal constructor" << endl;
11      }
12      ~Animal(){
13          cout << "Animal destructor" << endl;
14      }
15
16      void makeSound() const
17      {
18          cout << "Animal make sound" << endl;
19      }
20  };
```

```
22  class Dog:public Animal{
23  public:
24      Dog(){
25          cout << "Dog constructor" << endl;
26      }
27      ~Dog(){
28          cout << "Dog destructor" << endl;
29      }
30      void makeSound() const
31      {
32          cout << "Dog make sound" << endl;
33      }
34  };
```

```
36  class Cat:public Animal{
37  public:
38      Cat(){
39          cout << "Cat constructor" << endl;
40      }
41      ~Cat(){
42          cout << "Cat destructor" << endl;
43      }
44      void makeSound() const
45      {
46          cout << "Cat make sound" << endl;
47      }
48  };
```

在 main.cpp 程序中，定义一个通过基类 Animal&的传参来实现 makeSound 的调用。

```
 7  void func_r(const Animal& animal)
 8  {
 9      animal.makeSound();
10  }
```

```
24  int main()
25  {
26      Dog dog;
27      Cat cat;
28      func_r(dog);
29      func_r(cat);
30      return 0;
31  }
```

编译后运行的效果如下：

```
Animal constructor
Dog constructor
Animal constructor
Cat constructor
Animal make sound
Animal make sound
Cat destructor
Animal destructor
Dog destructor
Animal destructor
```

可以看到，派生类通过基类的引用传参后调用的 makeSound 并不是派生类中实现的功能，而是调用了基类中实现的函数功能，如何解决该问题，使用的技术便是多态的功能。

修改 animal.h 中的 Animal 基类，修改 makeSound()函数为虚函数，代码如下：

```
16      virtual void makeSound() const
17      {
18          cout << "Animal make sound" << endl;
19      }
20  };
```

执行效果如下：

```
Animal constructor
Dog constructor
Animal constructor
Cat constructor
Dog make sound
Cat make sound
Cat destructor
Animal destructor
Dog destructor
Animal destructor
```

将基类的同名函数 makeSound()修改成虚函数后，实现了派生类调用各自的成员函数。

基类的参数传递

在 main.cpp 中，通过基类 Animal 的引用传参可以实现多态的，通过基类的指针和值传送是否可以实现多态，定义另两种传参函数：

```
11  //OK polymorphism
12  void func_p(const Animal *animal)
13  {
14      animal->makeSound();
15  }
16
17  //can't polymorphism
18  void func_e(const Animal animal)
19  {
20      animal.makeSound();
21  }
```

经过测试发现通过值传递参数的形式无法实现多态的特性，通过引用和指针的形式可以实现多态的特性。

纯虚函数

在实际应用中，我们可能用不到通过基类定义的对象(Animal animal)，因此希望当使用基类创建对象时编译器可以提示错误信息，可以通过定义一个虚函数来实现该功能。
可以修改基类的 makeSound 函数(除构造函数外的其它成员函数)为纯虚函数，代码如下：

```
16      virtual void makeSound() const = 0;
```

此时 main.cpp 中如果有 Animal animal;的定义会提示如下错误信息：

```
error: cannot declare parameter 'animal' to be of abstract type 'Animal'
note:   because the following virtual functions are pure within 'Animal':
```

含有纯虚函数的类仍然可以通过类的指针进行定义(如 Animal *pcat 是正确的)。
除析构函数外，纯虚函数无需实现函数体。构造函数不能是虚函数。

基类指针定义派生类

通过基类指针定义派生类，在 main.cpp 中有如下代码：

```
 9    int main()
10    {
11        Animal *pcat = new Cat();
12        pcat -> makeSound();
13        delete pcat;
14        return 0;
15    }
```

基类中 makeSound 函数代码如下：

```
17        virtual void makeSound() const
18        {
19            cout << "Animal make sound" << endl;
20        }
```

需要注意的是：基类和派生类中的同名函数必须返回值相同，函数后面的 const 也必须同时有或者同时无，否则不能视为同名函数。
运行后的结果：

```
Animal constructor
Cat constructor
Cat make sound
Animal destructor
```

可以看出：运行后没有对 Cat 类进行析构，因为我们使用基类定义的指针变量，基类中无法调用到派生类中的析构函数，解决方法是将基类的析构函数定义成虚函数即可，代码如下：

```
    virtual ~Animal()
    {
        cout << "Animal destructor" << endl;
    }
```

修改后运行结果：

```
Animal constructor
Cat constructor
Cat make sound
Cat destructor
Animal destructor
```

多态案例：简单工厂模式

简单工厂模式有三个角色构成：
工厂类角色：该模式的核心，含义一定的商业逻辑和判断逻辑，根据逻辑不同产生具体的工厂产品也不同，如下面例子中的 Gardener 类；
抽象产品角色：它一般是具体产品继承的基类或者实现接口，例子中的 Fruit 类。
具体产品角色：工厂类所创建的对象就是此角色的实例，该类是抽象产品的派生类，如 Apple 和 Grape 类。

实现一个基类 Fruit 类，该类为接口类(纯虚函数)，派生类有 Apple 和 Grape，同时有一个 Gardener 类，该类实现操作 Fruit 中的所有派生类的成员函数。
fruit.h 代码如下：

```
 7  class Fruit{
 8  public:
 9
10      virtual ~Fruit() = 0;
11      virtual void plant() = 0;
12      virtual void grow() = 0;
13      virtual void harvest() = 0;
14  };
15
16
17  class Apple:public Fruit{
18  public:
19      Apple();
20      ~Apple();
21
22      void plant();
23      void grow();
24      void harvest();
25  };
26
27  class Grape:public Fruit{
28  public:
29      Grape();
30      ~Grape();
31
32      void plant();
33      void grow();
34      void harvest();
35  };
```

```
37  enum {
38      APPLE = 0,
39      GRAPE = 1
40  };
41
42
43  class Gardener{
44  public:
45      Gardener();
46      ~Gardener(){}
47      Fruit* getFruit(int);
48
49  private:
50      Apple *apple;
51      Grape *grape;
52  };
```

从上面的代码中可以看到，Fruit 类中都是纯虚函数，只提供接口，其派生类中实现了各种

的成员函数。Gardener 类中定义了两个派生类的私有成员指针，通过 getFruit()函数获取派生类对象的指针。

fruit.cpp 中实现了类成员函数的函数体：

```
4   Fruit::~Fruit()
5   {}
6
7   Apple::Apple()
8   {
9       cout << "Apple constructor" << endl;
10  }
11
12  Apple::~Apple()
13  {
14      cout << "Apple destructor" << endl;
15  }
16
17  void Apple::plant()
18  {
19      cout << "Apple plant" << endl;
20  }
21
22  void Apple::grow()
23  {
24      cout << "Apple grow" << endl;
25  }
26
27  void Apple::harvest()
28  {
29      cout << "Apple harvest" << endl;
30  }
```

```
33  Grape::Grape()
34  {
35      cout << "Grape constructor" << endl;
36  }
37
38  Grape::~Grape()
39  {
40      cout << "Grape destructor" << endl;
41  }
42
43  void Grape::plant()
44  {
45      cout << "Grape plant" << endl;
46  }
47
48  void Grape::grow()
49  {
50      cout << "Grape grow" << endl;
51  }
52
53  void Grape::harvest()
54  {
55      cout << "Grape harvest" << endl;
56  }
57
58  Gardener::Gardener()
59  {
60      apple = NULL;
61      grape = NULL;
62  }
```

```
66  Fruit* Gardener::getFruit(int type)
67  {
68      Fruit *fruit = NULL;
69      if(type == APPLE)
70      {
71          if(apple == NULL)
72          {
73              apple = new Apple();
74          }
75          fruit = apple;
76      }
77      else if(type == GRAPE)
78      {
79          if(grape == NULL)
80          {
81              grape = new Grape();
82          }
83          fruit = grape;
84      }
85      return fruit;
86
87  }
```

main.cpp 中，我们只需要通过 Gardener 类(该类扮演上帝的角色)和 Fruit 类接口即可操作派生类，下面是操作 APPLE 的代码：

```
 6  int main()
 7  {
 8      Gardener tom;
 9
10      Fruit *fruit = tom.getFruit(APPLE);
11      fruit->plant();
12      fruit->grow();
13      fruit->harvest();
14
15      return 0;
16  }
```

上面的例子实现了软件设计模式中的简单工厂。