## 作用域操作符::

作用是告诉编译器这个方法存在于何处，或者说是属于哪个对象。

## 构造器

* 1. 与方法的区别
     1. 构造器的名字和类名必须一样
     2. 创建类实例的时候，会自动调用
     3. 构造器永远不会返回任何值(连void都不用写)
     4. 如果没有定义，编译器会定义一个空的构造器

1. 构造对象数组

Car mycar[10];

Mycar[x].running;

## 析构器——与构造器相对应的析构器

* 1. 与构造器区别
     1. 构造器就是申请分配内存，析构器就是清理内存
     2. 名字一样，不过加了个~
     3. 也不返回值
     4. 不带参数的
     5. 在比较复杂的类里，析构器很重要，要及时清理内存，否则可能引起内存泄露
  2. 析构器就是虚方法

ClxBase \*pTest = new ClxDerived;

.....

delete pTest;

//基类的析构器必须是虚函数，不然调用不到子类的析构器！！！

为了基类的指针删除一个派生类的对象时，派生类的析构函数可以正确被调用

只有一个类被用来作为基类的时候，我们才把析构函数写成虚函数！

## this指针

* 1. 使用this的基本原则

如果代码不存在二义性隐患，就不必使用this指针！

* 1. 隐含的this指针

int main ()

{

Point point1(10,10);

point1.MovePoint(2,2);

//MovePoint函数的原型事实上应该是void MovePoint(Point \*this,int a,int b);

在类中它是隐含的，所以可以显式写成this->x......

* 1. this指针传的是对象的首地址

## 类的继承

* 1. 基类、父类、超类
  2. 写法

class SubClass:public SuperClass{}

* 1. 继承机制下的构造器和析构器
     1. 直接继承父类构造器

Animal::Animal(std::string theName){

name=theName;

}

Pig::Pig(std::string theName):Animal(theName){

}

赋值动作其实发生在Animal方法里面。

* + 1. 先是析构子类，然后再析构父类

## 访问控制

* 1. C++的访问级别

public 任何代码

protected 这个类本身和它的子类

private 只有类本身

* 1. 继承机制中的访问控制

- class Pig : public Animal{......}

public

继承的方法和属性的访问级别不发生任何改变-即public仍可以被所有代码访问，protected只能由基类的子类访问，private则只能由基类本身访问

protected

把基类的访问级别改为protected ，如果原来是public的话，这将使得这个子类外部的代码无法通过子类去访问基类中的public

## 友元关系

friend class \*\*\*\*

## 静态属性和方法

* 1. 好处
     1. 可以在没有创建任何对象的情况下调用有关方法
     2. 能够让有关的数据仍在该类的所有对象间共享
  2. 注意
     1. 静态成员是所有对象共享的，所以不能在静态方法里面访问非静态的元素
     2. 非静态方法可以访问类的静态成员，也可以访问类的非静态成员
     3. 静态方法无法访问this指针：

因为静态方法的内存分配，静态方法不是属于某个特定的对象的，是全体对象共享的

* + 1. 在使用静态属性的时候，要为它们分配内存。

private：

static int count;

....

int Pet::count = 0;//这一句，做了两件事：分配内存+初始化

* + 1. 静态成员的调用：

坚持使用

ClassName::methodName();

不适用

objectName.methodName();

## 虚方法

* 1. new和delete

int \*pointer = new int ;

\*pointer =110;

std::cout<<\*pointer;

delete pointer;

* 1. 问题引发

Pet \*cat = new cat();

Pet \*dog = new dog();

注意这里不是 Cat cat();而是以pet指针的形式实例化一个对象

new一个对象的时候，如果我们实例化的是子类对象，然后子类对象对基类方法进行了覆盖，这时候在运行时调用方法，调用的将是基类的方法不是覆盖的

为什么？因为编译器规定，new在程序运行的时候才分配内存(分配cat类型的指针给pet类型的指针)，不在编译时分配，编译器则认为，该方法调用基类的方法，这样才是执行起来最快的解决方案，而解决这个问题的关键，

就是对虚方法的提出。

虚方法可以让程序在运行的时候，能够选择应该是它本身的指针的方法

* 1. 声明方法：

在基类

virtual void play();

* 1. 虚方法的继承：一旦基类声明某方法是虚方法，那么在子类就不可能再声明为非虚方法
  2. 习惯
     1. 如果拿不准要不要声明为虚方法，那就虚方法好了
     2. 在基类声明所有的方法为虚方法，会让最终生成的代码速度变得慢一些
     3. 实现一个多层次的类继承关系的时候，最顶级的基类应该只有虚方法

析构器就是虚方法！

## 抽象方法（纯虚方法）

* 1. 语法：在声明一个虚方法的基础上，在原型的末尾加上‘=0’(告诉编译器不要在这个类里寻找这个方法的实现！)
  2. 与虚方法的区别
     1. 虚函数可以被直接使用，也可以被子类(sub class)重载以后以多态的形式调用，而纯虚函数必须在子类(sub class)中实现该函数才可以使用，因为纯虚函数在基类(base class)只有声明而没有定义。
     2. 虚函数和纯虚函数可以定义在同一个类(class)中，含有纯虚函数的类被称为抽象类(abstract class)，而只含有虚函数的类(class)不能被称为抽象类(abstract class)。
  3. 多态性(一个接口，多种方法)
     1. 编译时的多态性：通过重载实现，运行速度快
     2. 运行时的多态性：通过虚函数实现，高度灵活和抽象

## 运算符重载

1. 写法

#include <iostream>

class Complex

{

public:

Complex();

Complex(double r,double i);

Complex operator+(Complex &d);

void print();

private:

double real;

double imag;

};

Complex::Complex()

{

real = 0;

imag = 0;

}

Complex::Complex(double r,double i)

{

real = r;

imag = i;

}

Complex Complex::operator+(Complex &d)

{

Complex c;

c.real = real + d.real;

c.imag = imag + d.imag;

return c;

}

void Complex::print()

{

std::cout<<"("<<real<<", "<<imag<<"i)\n";

}

int main()

{

Complex c1(3,4),c2(5,-10),c3;

c3 = c1 + c2;//c1+c2解释为 c1.operator+c2

std::cout<<"c1 = ";

c1.print();

std::cout<<"c2 = ";

c2.print();

std::cout<<"c1 + c2 =";

c3.print();

return 0;

}

Complex Complex::operator+(Complex &c2)

{

return Complex(real+c2.real,image + c2.image);

}

1. 不允许重载的运算符
   1. 成员访问运算符 .
   2. 成员指针访问运算符 .\*
   3. 域运算符 ::
   4. 尺寸运算符 sizeof
   5. 条件运算符 ?:
2. 隐含的参数

实际上，运算符重载函数有两个参数，但由于重载函数是Complex类中的成员函数，有一个函数是隐含着的，运算符函数是用this指针隐含式地访问类对象的成员

return Complex(real+c2.real,image + c2.image);

return Complex(this->real+c2.real,this->image + c2.image);

return Complex(c1.real+c2.real,c2.image + c2.image);

1. 类友员函数
2. 重载<<操作符

#include <iostream>

class Complex

{

public:

Complex();

Complex(double r,double i);

Complex operator+(Complex &d);

void print();

private:

double real;

double imag;

friend std::ostream& operator<<(std::ostream& os,Complex f);

};

Complex::Complex()

{

real = 0;

imag = 0;

}

Complex::Complex(double r,double i)

{

real = r;

imag = i;

}

Complex Complex::operator+(Complex &d)

{

Complex c;

c.real = real + d.real;

c.imag = imag + d.imag;

return c;

}

void Complex::print()

{

std::cout<<"("<<real<<", "<<imag<<"i)\n";

}

std::ostream& operator<<(std::ostream& os,Complex f);

int main()

{

Complex c1(3,4),c2(5,-10),c3;

c3 = c1 + c2;

std::cout<<"c1 = "<<c1<<"\n";

//c1.print();

std::cout<<"c2 = "<<c2<<"\n";

//c2.print();

std::cout<<"c1 + c2 ="<<c3<<"\n";

//c3.print();

return 0;

}

std::ostream& operator<<(std::ostream& os,Complex f)

{

os<<"("<<f.real<<", "<<f.imag<<"i)";

return os;

}

## 多继承

多继承是指一个子类继承多个父类。多继承对父类的个数没有限制，继承方式可以是公共继承、保护继承和私有继承，

不写继承方式，默认是private继承



## 虚继承

虚继承：多个派生类保存相同基类的同名成员时，虽可以在不同的数据成员中分别存放不同的数据 ，但我们只需要相同的一份。

解决了多父类重复成员只保留一份的问题。

比如现在有一个沙发床，它既有床的属性又有沙发的属性，它们都有长宽高的属性，但是我们却只需要知道它的一个状态的属性。

它的一个长宽高能体现多个状态。

家具{

属性：长，宽，高；

}

沙发床：public 沙发，public 床{

属性：长，宽，高；

}

沙发：public 家具   ----》改为：沙发：virtual public 家具

{

属性：长，宽，高；

}

床：public 家具   ----》改为：床：virtual public 家具

{

属性：长，宽，高；

}