类和对象

2023年9月9日

20:28

C++面向对象的三大特性：封装、继承、多态

C++认为万事万物都可以成为对象，对象上有其属性和行为

封装：

封装的意义：

* 讲属性和行为作为一个整体，来表现生活中的事物
* 将属性和行为加以权限控制

案例：设计一个圆类，求圆的周长

类关键字：class

类中的属性和行为 我们统一成为 成员

属性 成员属性 成员变量

行为 成员函数 成员方法

访问权限：

公共权限---public 成员 类内可以访问 类外可以访问

保护权限---protected 成员 类内可以访问 类外不可以访问

私有权限---private 成员 类内可以访问 类外不可以访问

struct和class区别：默认的访问权限不同

* struct：默认权限为公共
* class ：默认权限为私有

成员属性设置为私有的优点

* 将所有成员属性设置为私有，可以自己控制读写权限
* 对于写权限，我们可以检测数据的有效性

构造函数与析构函数

语法：类名(){ …. }---与普通函数相仿

构造函数是在创建对象时，对其进行属性赋值（初始化）

析构函数是在对象销毁前，对其进行属性数值清空（清除内存）

注意：

* 构造函数可以有参数，析构函数不能有参数
* 两种函数都是自动调用，没有返回值，且不写void

构造函数按照参数分类：无参构造和有参构造

构造函数按照类型分类：普通构造和拷贝构造

例如：

有类class Persoin{…..};

无参构造：

Person(){…..} 即可

有参构造

Person(int a){……} 即可

普通构造：

除了拷贝构造之外的一切构造函数

拷贝构造：

Person(const Person &p1){}

注：p1是已经创建好的类，且要加const防止p1被修改，还要加上&引用

各种构造函数的调用-----括号法、显示法、隐式转换法

1.括号法：

Person p1； //表示调用无参构造函数---即是创建对象p1

Person p2(10); //表示调用有参构造函数，因为在括号中传递了值

Person p3(p2); //表示调用拷贝构造函数，因为传入参数是一个对象

2.显示法

Person p1; //表示调用无参构造函数---与括号法一致

Person p2=Person(10); //表示调用有参构造函数---多了Person类名，类似函数调用

Person p3=Person(p1); //表示调用拷贝构造函数---传入值为一个对象

类似Person(10); 表示声明一个匿名对象，编译器会在这一行代码执行完毕将其回收，因为无名，所以无法后续调用使用

3.隐式转换法

Person p1; //表示无参调用

Person p2=10; //有参调用----编译器转换为 Person p2=Person(10);

Person p3=p2; //拷贝调用-----编译器相应转换成显示法

注意事项

1.调用无参构造函数时，不要加()，即是语句Person p1;不要写成Person p1();

因为编译器会将其处理为一个函数的声明

例如

void func();

Person p1();

前者是函数的声明，后者遵循相同的规则也会被处理成函数声明。

2.不用利用拷贝构造函数初始化匿名对象，即是语句Person(p3);

编译器会报错---p3重定义

Person(p3); 编译器会处理为Person p3; （直接去掉括号，变成一个对象的声明）

拷贝构造函数的调用时机

* 使用一个已经创建完毕的对象来初始化一个新对象
* 值传递的方式给函数参数传值
* 以值方式返回局部对象

展示第二项和第三项的实例

有类class Person{……};

例如

void Test02(Person p)

{

}

Person Test03()

{

Person p1;

return p1;

}

int main()

{

Person p1；

Test02(p1); //此时调用函数时会相应的调用拷贝构造函数，涉及到值传递

Person p2=Test03(); //此时调用函数时也会调用拷贝构造函数，涉及到以值的方式返回局部对象

}

构造函数的调用规则

创建一个类，C++编译器会给这个类至少添加三个函数

1.默认构造函数（空实现）

2.默认析构函数（空实现）

3.默认拷贝构造函数（值拷贝）

* 如果用户定义有参构造函数，C++不再提供默认无参构造函数，但是会提供默认拷贝构造函数
* 如果用户定义拷贝构造函数，C++不会再提供其他构造函数

在第一项的情况下，调用默认构造函数，Person p1；编译器会报错

深浅拷贝

浅拷贝：简单的赋值拷贝操作

深拷贝：再堆区重新申请空间，进行拷贝操作

深拷贝代码

class Person

{

int m\_age;

int\* m\_Higheit;

}

Person(const Person &p)

{

m\_age=p.m\_age;

//m\_Higheit = p.m\_Higheit; //编译器自动生成的浅拷贝代码

m\_Higheit=new int(\*m\_Higheit); //用户子定义深拷贝代码

}

简单的值拷贝，如果拷贝指针，会将相同的地址进行拷贝，堆区内存使用完毕后应当用delete释放，但是两个对象拥有相同的地址时，第一个对象先释放内存，当第二个对象再使用这片空间时，就会存在非法访问的情况

如果属性有在堆区开辟的，一定要自己提供拷贝函数，防止浅拷贝带来的问题

初始化列表

例如

Person(int a,int b,int c):m\_A(a),m\_B(b),m\_C(c);

调用时直接

Person p1(10,20,30);即可

静态成员

静态成员变量

* 所有对象共享同一份数据
* 在编译阶段分配内存
* 类内声明，类外初始化

（静态成员变量也是有访问权想的，public private，protected）

注意：

静态成员变量不属于某一个对象，因为所有对象都共享同一份数据，因此静态成员变量有两种访问方式

1.通过对象进行访问

Person p1； 利用p1.静态成员，即可访问

2.通过类名进行访问

利用Person::静态成员，即可访问

静态成员函数

* 所有对象共享同一个函数
* 静态成员函数只能访问静态成员变量

访问方式和静态成员变量的访问一样，有两种

静态成员函数也是有访问权限的

成员变量和成员函数分开储存的

空对象占用内存空间为：1

C++编译器会给每个空对象分配一个字节空间，是为了区分空对象占内存的位置，且每个空对象也应该有一个独一无二的内存地址

静态成员变量不属于类对象上，非静态成员函数不属于类对象上，静态成员函数不属于类对象上

this指针：this指针是个指针常量（ ClassName \*const this ）

this指针指向被调用的成员函数所属的对象

this指针是隐含每一个非静态成员函数内的一种指针

this指针不需要定义，直接使用即可

this指针的用途

* 当形参和成员变量同名时，可以用this指针来区分
* 在类的非静态成员函数中返回对象本身，可以使用return \*this

关于this的程序

//创建类Person

class Person

{

public:

Person(int age)

{

age=age;

}

void Add(Person p)

{

this->age+=p.age;

}

int age;

}

void test01()

{

Person p1(18);

cout<<"p1的年龄为： "<<endl;

}

void test02()

{

Person p1(10);

Person p2(20);

//链式编程

p2.Add(p1).Add(p1).Add(p1);

cout<<"p2的年龄为："<<endl;

}

注意：

以上test01会报错，因为四个age无法区分，所以最终打印值是未知数，要防止重名的现象，应该利用this指针加以区分，修改语句为：this->age=age; 即可

以上test02也会报错，链式编程语句所调用函数返回值为void不能链接到后面的部分，所以需要修改test02函数，更换返回值，且需返回调用此函数的对象本身

函数修改如下：

Person& Add(Person p)

{

this->age=p.age;

return \*this; //返回调用此函数的对象本身

}

注意此处返回时应用了&引用，是为了保证返回的对象就是p2

Person Add(Person p)

{

this->age=p.age;

return \*this;

}

如果去掉引用，根据拷贝函数的值传递规则，编译器调用拷贝函数，创建新的空间将\*this(p2本身)拷贝到这片新的空间，所以此后再次链接时就不是p2本身在链接后续函数了，而是其他空间的对象在进行操作，但是值一直在继承，最终链式编程语句结束的结果为40，也就是进行了3次加法

空指针也可以访问类成员函数，但是记住一定不能访问成员变量，因为传入的空指针被this拷贝（不知道说法对不对，如果p是空，this也将是空指针），在进行变量赋值时，

比如 m\_Age=age;

实际是 this->m\_Age=age;

因为this是空指针，所以空指针访问成员变量就会报错

const修饰成员函数

* 常函数内不可以修改成员属性

在成员函数后面加const，本质是在this指针前加上了const，限定其值不能修改

* 成员属性声明时加关键字mutable后，在常函数中依然可以修改

mutable可以理解为取消const限制

常对象

常对象只能调用常函数

常对象不可以调用普通成员函数，因为普通成员函数可以修改成员属性

class GoodGay;

class Building;

友元friend

全局函数做友元 friend void func();

类做友元 friend class 类名

成员函数做友元 friend void GoodGay::visit();