**C++学习总结**

# 内存分区模型

## 代码区

### 存放函数的二级制代码，由操作系统进行管理的

## 全局区

### 存放全局变量和静态变量以及常量

## 栈区

### 由编译器自动分配释放，存放函数的参数值，局部变量等

## 堆区

### 由编译器自动分配释放，存放函数的参数值，局部变量等

## 内存四区的意义

### 不同区域存放的数据，赋予不同的声明周期，给我们更大的灵活编程

## 程序运行前

### 代码区

### 全局区

* 全局变量和静态变量存放于此
* 全局区还包含了常量区，字符串常量和其他常量也存放于此
* 该区域的数据在程序结束之后由操作系统释放

## 程序运行后

### 栈区

* 由编译器自动分配释放，存放函数的参数值，局部变量等。
* 注意事项：不要返回局部变量的地址，栈区开辟的数据由编译器自动释放

### 堆区

* 有程序员分配释放，若程序员不释放，程序结束之后有操作系统回收
* 在C++中主要利用new在堆区中开辟内存

## new运算符

### 在堆区开辟数据

### 堆区开辟的数据，由程序员手动开辟，手动释放，释放用delete

### 示例用法

* 创建

• int\* p = new int(10);//分配一个整型，值为10，p指向它

• int\* arry = new int[10];//分配一个人10个int的数组;p指向第一个int

* 删除

• delete p;//p必须指向一个动态分配的对象或为空

• delete[] arry;//arry必须指向一个动态分配的数组或为空

# 引用

## 作用

### 给变量起别名

## 语法：数据类型 &别名 = 原名

### int a = 0;

### int &b = a;

### //a和b操作的是同一块内存

## 注意事项

### 引用必须初始化——告诉它它是谁的别名

### 引用在初始化之后，不可以改变

## 常用用法

### 做函数参数

* 作用

• 函数传参时，可以利用引用让形参修饰实参

* 优点

• 可以简化指针修改实参（传址）

* 示例

• void changeNums(int &a,int& b);//引用传递int main(void){ int a = 10; int b = 20; changeNums(a,b); //引用——其实上面的a就是下面a的一个别名 return 0;}

## 引用本质

### 引用的本质在c++内部实现是一个指针常量,引用一旦被初始化之后就不能更改。

## 结论

### C++推荐使用引用技术，因为语法方便，引用本质是指针常量，但所有的指针操作编译器都棒我们做了。

# 函数提高

## 函数默认参数

### 在c++中函数形参列表中的形参是可以有默认值的。

### 语法

* 返回值类型 函数名(参数=默认值)

### 案例

* //函数的哪个参数被声明默认了，下面函数调用的时候就可以少传哪个参数，如果有默认值还传了参数，用的就是函数调用传递的参数int func(int a,int b =10,int c =23){ return a+b+c;}int main(void){ int ref = func(10); return 0;}

### 注意事项

* 如果某个位置已经有了默认参数，那么从这个位置往后都要有默认参数

### 案例

* //从b开始往后一的参数都有默认参数int fun2(int a,int b= 10;int c =20){ }

### 注意事项

* 如果函数的声明有默认参数，函数的实现就不能有默认参数了。
* 声明和实现只能有一个有默认参数。

### 案例

* int fun3(int a = 10;int b = 20);int fun3(int a,int b){ }

## 函数占位参数

### C++中函数的形参列表里可以有占位参数，用来占位，调用函数的时候必须填补该位置。

### 语法

* 返回值类型 函数名 （数据类型）{}，

### 案例

* void func(int a, int){}int main(void){func(10,1);//这个1传进去是拿不到的，目前阶段的占位参数我们还用不到，但在后面是会用到的。return 0;}

### 占位参数还可以有默认参数

## 函数重载

### 作用

* 函数名可以相同，提高复用性

### 函数重载满足条件

* 同一个作用域下
* 函数名相同
* 函数参数类型不同或者个数不同或者顺序不同

### 注意

* 函数的返回值不可以作为函数重载的条件

### 案例

* void func(){cout << "无参数" << endl;}void func(int a){cout << a;}int main(void){ 根据函数传递参数的不同调用不同的代码func();func(10);return 0;}

### 注意事项

* 引用作为函数重载条件
* 函数重载碰到函数默认参数

### 案例

* #include<iostream>using namespace std;void func(int &a){//int& a =10;不合法}void func(const int &a){//const int& a =10;合法——编译器自动优化}int main(void){func(10);return 0;}
* #include<iostream>using namespace std;void func(int a ,int b = 10){}void func(int a){}int main(void){func(10);/\*当函数重载碰到默认参数编译器傻了，不知道该调用哪个了——出现二义性——写函数重载就不要加默认参数，避免这种情况的出现\*/return 0;}

# 类和对象

## 三大特性

### 封装

### 继承

### 多态

## 封装

### 意义

* 将属性和行为作为一个整体，表现生活中的事物

• 语法

• class 类名{访问权限: 属性 / 行为};

• 案例

• 创建一个圆类，求圆的周长

• #include<iostream>using namespace std;double pi = 3.14;//class 代表设计一个类，类后面紧跟着的就是类名称class Circle{//访问权限//公共权限public://属性 //半径int c\_r;//行为//获取圆的周长double calculateZC(){return 2 \* pi \* c\_r;}};int main(void){//通过圆类创建具体的圆(对象)//实例化——通过一个类创建一个对象的过程Circle c1;//给圆对象的属性进行赋值c1.c\_r = 10;cout << "圆的周长为" << c1.calculateZC() << endl;return 0;}

• 创建一个学生类

• #include<iostream>#include<string>using namespace std;class Student{public:string s\_Name;int s\_Id;void showStudent(){cout << "姓名： " << s\_Name << "ID：" << s\_Id << endl;}//赋值void inputName(string name){s\_Name = name;}};int main(void){Student s1;//s1.s\_Name = "张三";s1.inputName("赵六");s1.s\_Id = 123456;s1.showStudent();return 0;}

* 将属性和行为加以权限控制

• 类在设计时，可以把属性和行为放在不同的权限下，加以控制

• 访问权限有三种

• public

• 公共权限——成员类内可以访问，类外可以访问

• protected

• 保护权限——成员类内可以访问，类外不可以访问

• private

• 私有权限——成员类内可以访问，类外不可以访问

• 案例

• #include<iostream>#include<string>using namespace std;class Person{public:string p\_name;protected:string p\_car;private:int p\_password;public:void funcshow(){p\_name = "张三";p\_car = "拖拉机";p\_password = 123456;}};int main(void){Person p1;p1.p\_name = "王五";//p1.p\_car = "GTR";protected类外无法访问//p1.p\_password = 123;private类外无法访问return 0;}

## 通俗的例子

### 人可以作为对象，属性有姓名、年龄、身高、体重…行为有唱、跳、跑…

### 车也可以作为对象，属性有轮胎、方向盘、大灯…行为有载人、放音乐、开空调…

### 具有相同性质的对象，我们可以抽象称为类，人属于人类，车属于车类…

## 类中的属性和行为，我们统称为成员

### 属性-成员属性-成员变量

### 行为-成员函数-成员方法

## struct和class的区别

### 在C++中struct和class的唯一区别就是默认的访问权限不同。

### 区别

* struct默认权限为公共public
* class默认权限为私有private

### 成员属性设置为私有

* 优点1：将所有成员属性设置为私有，可以自己控制读写权限。
* 优点2：对于写权限，我们可以检测数据的有效性。

## 实际编程案例

### 设计一个圆类和一个点类判断圆和点的关系。

### 提示

* 在一个类中可以让另一个类作为这个类的成员

### #include<iostream>using namespace std;class Point{public:void setx(int x){c\_x = x;}int getx(){return c\_x;}void sety(int y){c\_y = y;}int gety(){return c\_y;}//建议将属性设置为私有，对外提供接口private:int c\_x;int c\_y;};class Circle{public:void setr(int r){c\_R = r;}int getr(){return c\_R;}void setcenter(Point center){c\_center = center;}Point getcenter(){return c\_center;}private:int c\_R;Point c\_center;};//判断void isInCircle(Circle &c,Point &p){int distance =(c.getcenter().getx() - p.getx()) \* (c.getcenter().getx() - p.getx()) +(c.getcenter().gety() - p.gety()) \* (c.getcenter().gety() - p.gety());int rdistance = c.getr() \* c.getr();if (distance == rdistance){cout << "点在圆上" << endl;}else if (distance > rdistance){cout << "点在圆外" << endl;}else{cout << "点在圆内" << endl;}}int main(void){Circle c1;c1.setr(10);Point center;center.setx(10);center.sety(10);c1.setcenter(center);Point p1;p1.setx(3);p1.sety(4);//调用判断isInCircle(c1, p1);return 0;}

## 将一个类拆分成两个文件的案例

### point.h

* #pragma once#include<iostream>using namespace std;class Point{public:void setx(int x);int getx();void sety(int y);int gety();private:int c\_x;int c\_y;};

### point.cpp

* #include"point.h"//Point::告诉编译器这是Point作用域下面的一个成员函数void Point::setx(int x){c\_x = x;}int Point::getx(){return c\_x;}void Point::sety(int y){Point::c\_y = y;}int Point::gety(){return c\_y;}

## 对象的初始化与清理

### 意义

* 在生活中我们所购买的点子产品大多都有恢复出厂设置，在某一天我们不使用的时候清楚自己的数据来保证自己信息的安全。
* C++中的面向对象来源生活，每个对象也会有初识设置以及对象销毁前的清理数据的设置。
* 对象的初始化和清理也是两个非常重要的安全问题。
* 一个对象或者变量没有初识状态，对其使用后的后果是未知的。
* 同样的使用完一个对象或者变量，没有及时进行清理，也会造成一定的安全问题。
* C++利用了构造函数和析构函数解决上述问题，这两个函数将会被编译器自动斓用，完成对象初始化和清理工作。对象的初始化和清理工作是编译器强制要我们做的事情，因此如果我们不提供构造和析构，编译器会提供，但是编译器提供的构造函数和析构函数是空实现。

### 构造函数

* 作用

• 主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用，无须手动调用。

* 语法

• 类名(){}

* 写法

• 构造函数没有返回值也不写void

• 函数名称与类名相同

• 构造函数可以有参数，因此可以发生重载

• 程序在调用对象的时候会自动调用构造，无须手动调用，而且只会调用一次

* 案例

• #include<iostream>using namespace std;class Person{public:Person(){ //不写的也会自动创建一个，只不过里面是空的cout << "构造函数的调用" << endl;}};void test01(){Person p;//创建了一个对象但是没有调用这个函数}int main(void){test01(); system("pause");return 0;}

### 构造函数的分类及调用

* 两种分类方式

• 按参数分为:有参构造和无参构造

• 有参构造

• 调用构造函数时必须传递参数，也就是构造函数的括号里面必须有东西

• 无参构造

• 构造函数的括号里不能有东西

• 按类型分为:普通构造和拷贝构造

• 普通构造

• 除了拷贝构造之外的就是普通构造

• 拷贝构造

• 当调用一个类的构造函数的时候传递一个同样的类进去，并复制传递进去的类构造出一个相同的类

* 三种调用方式

• 括号法

• 案例

• Person p;//默认构造函数调用/\*注意：使用默认构造函数的时候，不要加(),编译器会认为这是一个函数的声明例如：Person p1();不会认为在创建对象\*/Person p2(10);//有参构造函数调用Person p3(p2);//拷贝构造函数调用cout << "p2的年龄为" << p2.age << endl;cout << "p3的年龄为" << p3.age << endl;

• 显示法

• 案例

• Person p1;//无参Person p2 = Person(10);//有参Person p3 = Person(p2);//拷贝//如果把等号右边的式子单独拿出来//Person(10)这是一个匿名对象-特点——当前行执行结束后，系统会立即回收掉匿名对象//注意：不要利用拷贝函数初始化匿名对象-编译器会认为Person(p3) == Person p3 编译器会认为是对象的声明//Person(p3)

• 隐式转换法

• 案例

• Person p4 = 10;//相当与Person p4 = Person(10);Person p5 = p4;//拷贝构造

* 构造函数的调用规则

• 默认情况下，C++编译器至少给一个类添加三个函数

• 默认构造函数(无参、函数体为空)

• 默认析构函数(无参、函数体为空)

• 默认拷贝函数构造函数，对属性值拷贝

• 构造函数调用规则

• 如果用户定义有参构造函数，C++不再提供默认无参构造，但是会提供默认拷贝构造

• 如果用户定义拷贝构造函数，C++不会再提供其他构造函数

• 案例

• #include<iostream>using namespace std;//构造函数的调用规则//只要创建一个类，c++编译器会默认给每个类都添加至少3个函数/\*默认构造(空实现)析构函数(空实现)拷贝函数\*/class Person{public:Person(){cout << "Person的默认构造函数调用" << endl;}Person(int age){m\_Age = age;cout << "Person的有参构造函数调用" << endl;}Person(const Person& p){m\_Age = p.m\_Age;cout << "Person的拷贝构造函数调用" << endl;}~Person(){cout << "Person的默认析构函数调用" << endl;}int m\_Age;};void test(){Person p;p.m\_Age = 18;Person p2(p);cout << "p2的年龄为" << p2.m\_Age << endl;}//当用户创建了有参构造函数，编译器就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void test02(){}int main(void){test02();system("pause");return 0;}

### 析构函数

* 作用

• 主要作用在于对象销毁前系统自动调用，执行一些清理工作。

* 语法

• ~类名(){}

* 写法

• 析构函数没有返回值也不写void

• 函数名称与类名相同，在名称前加上~

• 析构函数不可以有参数，因此不可以发生重载

• 程序在对象销毁前会自动调用析构，无须手动调用，而且只会调用一次

* 案例

• #include<iostream>using namespace std;class Person{public:Person(){cout << "构造函数的调用" << endl;}~Person(){cout << "析构函数的调用" << endl;}//构造和析构都是必须有的实现，如果我们自己不提供，编译器会提供一个空实现的构造和析构};void test01(){Person p;//在栈上的数据，test01执行完之后会释放这个对象}int main(void){test01();//Person p;在main函数中析构函数也会被调用在按完任意键之后system("pause");return 0;}

### 总结

* 用户提供了有参，编译器不会提供无参，但会提供拷贝；
* 用户提供了拷贝，编译器什么构造函数都不会提供。

### 深拷贝与浅拷贝（重中之重）

* 浅拷贝

• 定义

• 简单的赋值拷贝操作。

• 问题

• 内存重复释放。

• 案例

• #include<iostream>using namespace std;//深拷贝与浅拷贝问题class Person{public:Person(){cout << "Person的默认构造函数调用" << endl;}Person(int age,int height){m\_Height = new int(height);m\_Age = age;cout << "Person的有参构造函数调用" << endl;} Person(const Person& p){cout << "Person的拷贝构造函数调用" << endl;m\_Age = p.m\_Age;m\_Height = p.m\_Height;编译器默认实现的就是这行代码}~Person(){//将堆区开辟的数据进行释放if (m\_Height !=NULL){delete m\_Height;m\_Height = NULL;}cout << "Person的析构构造函数调用" << endl;} int m\_Age;int\* m\_Height;//为什么要用指针——要把身高开辟到堆区};void test(){Person p1(18,166);cout << p1.m\_Age<<"\t" << \*p1.m\_Height << endl;Person p2(p1);cout << p2.m\_Age<<"\t" <<\*p2.m\_Height<< endl;}int main(void){test();system("pause");return 0;}

• 解决方案

• 浅拷贝的这个问题需要用深拷贝来解决

• 重新在堆区找一块内存来存放他。

• 自己实现拷贝构造函数来解决浅拷贝带来的问题

* 深拷贝

• 定义

• 在堆区中重新申请空间，进行拷贝操作。

• 解决上述问题的具体方法

• Person(const Person& p){cout << "Person的拷贝构造函数调用" << endl;m\_Age = p.m\_Age;//m\_Height = p.m\_Height;编译器默认实现的就是这行代码//深拷贝操作m\_Height = new int(\*p.m\_Height);}

* 总结

• 如果有属性在堆区开辟的，一定要自己提供拷贝构造函数，防止浅拷贝带来的问题。

### 初始化列表

* 作用

• C++提供了初始化列表语法，用来初始化对象。

* 语法

• 构造函数()：属性1（值1），属性2（值2）…{}

* 案例

• #include<iostream>using namespace std;class Person{public://传统赋值操作/\*Person(int a, int b, int c){m\_A = a;m\_B = b;m\_C = c;}\*///初始化列表初始化属性Person(int a,int b,int c) :m\_A(a), m\_B(b), m\_C(c){}int m\_A;int m\_B;int m\_C;};void test(){//Person p(10,20,30);Person p(30,20,10);cout << p.m\_A << endl;cout << p.m\_B << endl;cout << p.m\_C << endl;}int main(void){test();system("pause");return 0;}

### 类对象作为类成员

* 定义

• C++中类的成员可以是另一个类的对象，我们称该成员为对象成员。

* 案例

• class A{}class B{ A a;}

• B类中有对象A作为成员，A为对象成员。

* 构造顺序

• A先被构造

• 当其他类的对象作为本类的成员时，构造时先构造其他类的对象，再构造自身。

• 析构呢？与构造函数相反。

• 自身的析构函数先进行，之后其它类再进行。

* 案例（可以自己试试跑）

• #include<iostream>#include<string>using namespace std;class Phone{public:Phone(string p){Phonename = p;cout << "Phone的构造函数调用" << endl;}~Phone(){cout << "Phone的析构函数调用" << endl;}string Phonename;};class Person{public://Phone Personphone = pname 隐式转换法Person(string name, string pname):Personname(name), Personphone(pname){cout << "Person的构造函数调用" << endl;}~Person(){cout << "Person的析构函数调用" << endl;}string Personname;Phone Personphone;};void test(){Person p("张三", "华为"); cout << p.Personname<< endl;cout << p.Personphone.Phonename<< endl;}int main(void){test();system("pause");return 0;}

### 静态成员

* 定义

• 静态成员就是在成员变量和成员函数前面加上关键字啊static，称为静态成员。

* 分类

• 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

• 函数

• 所有成员共享同一个函数

• 静态成员函数只能访问静态成员变量

## C++对象模型和this指针

### C++对象模型

* 成员变量和成员函数分开存储
* 在C++中，类内的成员变量和成员函数分开存储，
* 只有非静态成员变量才属于类的对象上。
* （只有非静态成员变量的大小算进类的大小中，其他的都不算。）
* 空对象的大小是1，为的是区分不同类在内存中的占用位置。

### this指针

* 概念

• 通过上一个知识点《成员变量和成员函数是分开存储的》我们知道C++中成员变量和成员函数是分开存储的。

• 每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会公用一块代码。

• 那么问题是：这一块代码是如何区分是哪个对象调用自己的呢？

• C++通过提供特殊的对象指针，this指针，解决上述问题。

• this指针指向被调用的成员函数所属的对象。

• (谁调的，this就指向谁)

• this指针是隐含每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。

* 用途

• 当形参和成员变量同名时，可用this指针来区分

• 在类的非静态成员函数中返回对象本身，可使用return \*this

* 案例

• 解决名称冲突，返回对象本身用\*this

• #include<iostream>using namespace std;class Person{public:Person(int age){//this指针指向的是被调函数的成员函数所属的对象//这里指向的就是pthis->age = age;}//返回本体要用应用的方式进行返回//这里返回值如果是Person，就创建了一个新的对象Person& PersonAddPerson(Person &p){this->age += p.age;return \*this;}int age;//注意起名规范也可以解决名字冲突的问题};//解决对象冲突void test(){Person p(18);cout << p.age << endl;}//返回对象本身用\*thisvoid test01(){Person p1(10);Person p2(10);p2.PersonAddPerson(p1);//将p1和p2的加在一起//多次追加,return \*this;//链式编程思想p2.PersonAddPerson(p1).PersonAddPerson(p1);cout << p2.age << endl;}int main(void){test01();system("pause");return 0;}

### const修饰成员函数

* 常函数

• 成员函数后加const后我们称这个函数为常函数

• 常函数不可以修改成员属性

• 成员属性声明时加关键字mutable后，在常函数中依然可以修改

* 常对象

• 声明对象前const称该对象为常对象。

• 常对象只能调用常函数。

* 案例

• #include<iostream>using namespace std;//常函数class Person{public://this指针的本质是指针常量，指针的指向是不可以修改的//就相当于Person \*const this;//在成员函数后面加const修饰的是this指向，让指针指向的值也不可以修改void showPerson() const//加个const就不允许修改了{this->m\_b = 100;//this = NULL;tbhis指针是不可以修改指针的指向的}int m\_a;mutable int m\_b;//加了mutable修饰的特殊变量，即使在常函数,常对象中，也可以修改这个值void func(){m\_a = 100;//在普通成员函数中是可以修改的}};void test(){Person P;P.showPerson();}//常对象void test1(){const Person p;//在对象前加const，变为常对象//p.m\_a = 100;p.m\_b = 100;//常对象只能调用常函数 p.showPerson();//p.func();常对象不能调用普通成员函数，因为普通成员函数可以修改属性。}int main(void){test();system("pause");return 0;}

## 友元

### 概念

* 客厅就是Public，你的卧室就是Private
* 客厅所有人都可以进去，但是你的卧室只有和你亲密的人可以进。
* 在程序中，有些私有属性也想让类外特殊的一些函数或者类进行访问，就需要用到友元技术。
* 友元的目的就是让一个函数或者类 访问另一个类中的私有元素。

### 关键字

* friend

### 实现方式

* 全局函数做友元

• 就是将此函数在类的最上面写一个声明，前面加一个friend。

• #include<iostream>#include<string>using namespace std;class Building {//goodgay全局函数是Building类的一个好朋友，可以访问你家的卧室(私有成员)friend void goodgay(Building\* building);public:Building(){m\_SittingRoom = "客厅";m\_BedRoom = "卧室";}public:string m\_SittingRoom;private:string m\_BedRoom;};//全局函数void goodgay(Building\* building){cout << "好基友全局函数正在访问你的" << building->m\_SittingRoom << endl;cout << "好基友全局函数正在访问你的" << building->m\_BedRoom << endl;}void test(){Building building;goodgay(&building);}int main(void){test();system("pause");return 0;}

* 类做友元

• 一个类在另一个中friend class xx。

• #include<iostream>#include<string>using namespace std;//在前面先声明一下class Building;class GoodGay{public:GoodGay();public:void visit();//参观函数 访问Building中的属性Building\* building;};class Building{//GoodGay是Building类的好朋友，可以访问其私有属性friend class GoodGay;public:Building();public:string m\_SittingRoom;private:string m\_BedRoom;};//在类外写成员函数Building::Building(){m\_SittingRoom = "客厅";m\_BedRoom = "卧室";}GoodGay::GoodGay(){//创建一个Building对象building = new Building;}void GoodGay::visit(){cout << "好基友正在访问你的" << building->m\_SittingRoom << endl;cout << "好基友正在访问你的" << building->m\_BedRoom << endl;}void test(){GoodGay gy;gy.visit();}int main(void){test();system("pause");return 0;}

* 成员函数做友元

• 告诉编译器 另一个类中的xx成员函数作为本类的好朋友，可以访问私有函数。

• #include<iostream>#include<string>using namespace std;class Building;class GoodGay{public:GoodGay();void visit();//可以访问Building中私有成员void visit1();//不可以访问Building中私有成员Building\* builidng;};class Building{//告诉编译器 GoodGay类中的visit成员函数作为本类的好朋友，可以访问私有函数friend void GoodGay::visit();public:Building(); public:string m\_SittingRoom;private:string m\_BedRoom;};Building::Building(){m\_SittingRoom = "客厅";m\_BedRoom = "卧室";}GoodGay::GoodGay(){builidng = new Building;}void GoodGay::visit(){cout << "visit正在访问" << builidng->m\_SittingRoom << endl;cout << "visit正在访问" << builidng->m\_BedRoom << endl;}void GoodGay::visit1(){cout << "visit1正在访问" << builidng->m\_SittingRoom << endl;}void test(){GoodGay gg;gg.visit();gg.visit1();}int main(void){test();system("pause");return 0;}

## 运算符重载

### 概念

* :对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

### 分类

* 加号运算符重载

• 作用

• 实现两个自定义数据类型相加的运算。

• 例如：两个整型相加编译器知道该怎么进行运算，如果是两个自定义出来的类型，两个Person想加，编译器就不知道该怎么运算了。

• 案例

• #include<iostream>#include<string>using namespace std;//加号运算符重载class Person{public://1.成员函数重载+/\*Person operator+(Person& p){Person temp;temp.m\_A = this->m\_A + p.m\_A;temp.m\_B = this->m\_B + p.m\_B;return temp;}\*/int m\_A;int m\_B;};//2.全局函数重载+Person operator+(Person& p1, Person& p2){Person temp;temp.m\_A = p1.m\_A + p2.m\_A;temp.m\_B = p1.m\_B + p2.m\_B;return temp;}//函数函数重载版本Person operator+(Person& p1, int num){Person temp;temp.m\_A = p1.m\_A + num;temp.m\_B = p1.m\_B + num;return temp;}void test01(){Person p1;p1.m\_A = 10;p1.m\_B = 10;Person p2;p2.m\_A = 10;p2.m\_B = 10;//成员函数重载本质调用//Person p3 = p1.operator+(p2);//Person p3 = p1 + p2;//可以简化成这种形式//全局函数重载的本质调用//Person p3 = operator+(p1,p2);/\*cout << p3.m\_A << endl;cout << p3.m\_B << endl;\*///运算符重载也可以发生函数重载Person p3 = p1 + 10;cout << p3.m\_A << endl;cout << p3.m\_B << endl;}int main(void){{test01();system("pause");return 0;}

• 总结

• 对于内置的数据类型的表达式的运算符是不可能改变的

• 不要滥用运算符重载

* 左移运算符重载（<）

• 作用

• 可以输出自定义的类型

• 在C++中<<一般用在cout中，但是类不属于可以输出的类型，所以本来在cout中的<<后面一般是不能直接加类这个整体的，但是我们在重载后可以做到

• 案例

• #include<iostream>using namespace std;class Person{friend ostream& operator<<(ostream& cout, Person& p);public:Person(int a, int b){m\_A = a;m\_B = b;}//利用成员函数重载左移运算符p.operator<<(cout)简化版本p<<cout//一般我们不会利用成员函数来重载<<运算符，以为无法实现cout在左边/\*void operator<<(ostream &cout,Person &p){cout << p.m\_A << endl;cout << p.m\_B << endl;}\*/private:int m\_A;int m\_B;};//只能利用全局函数来重载左移运算符ostream& operator<<(ostream &cout, Person &p) //这样写的本质就是operator<<(cout,p)简化版本就是cout<<p; {cout << p.m\_A << endl;cout << p.m\_B << endl;return cout;}void test(){Person p(10,10);cout << p << "hello world" << endl;}int main(void){test();system("pause");return 0;}

• 总结

• 重载左移运算符配合友元可以实现输出自定义数据类型。

• 另一种实现方式

• void operator<<(ostream &cout){cout << this->m\_A;cout << this->m\_B;}//使用p<<cout;

* 递增运算符重载（++）

• 作用

• 通过重载递增运算符，实现自己的整型数据。

• 案例

• #include<iostream>using namespace std;//重载递增运算符class MyInteger{friend ostream& operator<<(ostream& cout, MyInteger myint);public:MyInteger(){m\_Num = 0;}//重载++运算符——前置//返回引用是为了一直对一个数据进行递增操作MyInteger& operator++(){++m\_Num;return \*this;}//重载++运算符——后置MyInteger operator++(int)//这个int在这里作为占位参数，用来区分前置递增和后置递增{MyInteger temp = \*this;m\_Num++;return temp;//后置递增要返回值，因为如果返回引用，这里相当于返回的是一个局部对象的引用。//局部对象在当前函数执行完毕之后就被释放掉了，还要返回引用就是非法操作。}private:int m\_Num;};//全局函数重载左移运算符ostream& operator<<(ostream& cout, MyInteger myint){cout << myint.m\_Num << endl;return cout; }void test(){MyInteger myint;cout << ++(++myint);cout <<myint;}void test02(){MyInteger myint;cout << myint++ << endl;cout << myint << endl;}int main(void){//test();test02();system("pause");return 0;}

• 总结

• 前置递增返回引用，后置递增返回值。

* 赋值运算符重载（=）

• 作用

• 如果我们想重定义赋值符号，例如Class A=Class B不是整个B复制到A中，而是只有B中的学号复制到A中，这时候就需要重定义赋值运算符

• C++编译器至少给一个类添加4个函数(前三个之前已经讲过了)

• 默认构造函数(无参，函数体为空)

• 默认析构函数(无参，函数体为空)

• 默认拷贝构造函数，对属性进行值拷贝

• 赋值运算符operator=，对属性进行值拷贝

• 如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题。

• 案例

• #include<iostream>using namespace std;class Person{public:Person(int age){m\_Age = new int(age);}~Person(){if (m\_Age != NULL){delete m\_Age;m\_Age = NULL;}}//重载赋值运算符Person& operator=(Person &p){//编译器默认提供的是浅拷贝操作//m\_Age = p.m\_Age;//应该先判断是否有属性在堆区，如果有先释放干净，然后再深拷贝。if (m\_Age != NULL){delete m\_Age;m\_Age = NULL;}//深拷贝操作m\_Age = new int(\*p.m\_Age);return \*this;}int \*m\_Age;};void test1(){Person p1(18);Person p2(20);Person p3(30);p3 = p2 = p1;cout << \*(p1.m\_Age) << endl;cout << \*(p2.m\_Age) << endl;cout << \*(p3.m\_Age) << endl;}int main(void){test1();system("pause");return 0;}

* 关系运算符重载

• 作用

• 重载关系运算符，可以让两个自定义类型对象进行对比操作

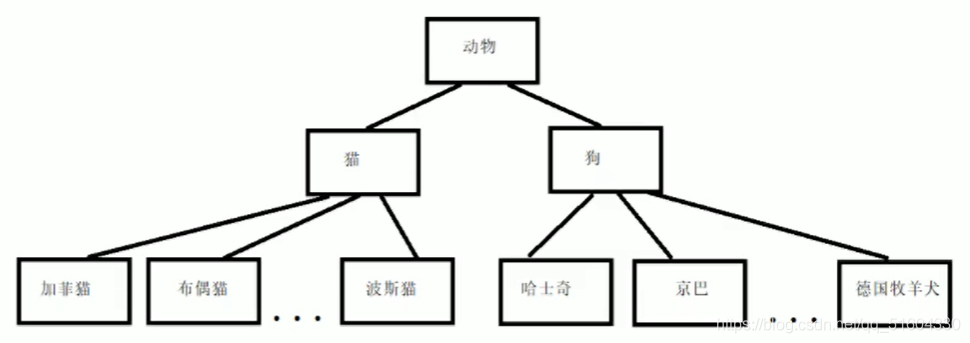
• 案例

• #include<iostream>#include<string>using namespace std;class Person{public://重载==bool operator==(Person &p){if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age){return true;}else{return false;}}bool operator!=(Person &p){if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age){return false;}else{return true;}}Person(string name, int age){m\_Name = name;m\_Age = age;}string m\_Name;int m\_Age;};void test(){Person p1("张三", 20);Person p2("张三", 20);if (p1 == p2){cout << "p1和p2是相等的" << endl;}else{cout << "p1和p2是不相等的" << endl;}if (p1 != p2){cout << "p1和p2是不相等的" << endl;}else{cout << "p1和p2是相等的" << endl;}}int main(void){test();system("pause");return 0;}

## 继承

### 引入

* 我们知道猫和老虎都属于猫科动物，狗和猫都属于动物，他们都有一部分相同的属性例如年龄，大小，体重，名字等，但是他们属于不同的类。所以当我们写这些类的时候就要重复编写这些属性或是方法，这时候会大大影响代码编写的效率，所以我们引入了继承，当遇到这种情况我们不需要重复编写代码



### 基本语法

* 例子

• 例如我们看到很多网站中，都有公共的头部，公共的底部，甚至公共的左侧列表，只有中心内容不同。

• 接下里我们分别利用普通写法和继承写法来实现网页中的内容，看一下继承存在的意义以及好处。

* 对比

• 不使用继承的方法

• #include<iostream>#include<string>using namespace std;//普通实现页面//java页面class Java{public:void header(){cout << "首页、登录注册" << endl;}void footer(){cout << "帮助中心、交流合作" << endl;}void left(){cout << "java、python、c++" << endl;}void contenet(){cout << "java学科视频" << endl;}};class Python{public:void header(){cout << "首页、登录注册" << endl;}void footer(){cout << "帮助中心、交流合作" << endl;}void left(){cout << "java、python、c++" << endl;}void contenet(){cout << "python学科视频" << endl;}};class Cpp{public:void header(){cout << "首页、登录注册" << endl;}void footer(){cout << "帮助中心、交流合作" << endl;}void left(){cout << "java、python、c++" << endl;}void contenet(){cout << "c++学科视频" << endl;}};void test(){cout << "java" << endl;Java java;java.header();java.footer();java.left();java.contenet();cout << endl;cout << "python" << endl;Python python;python.header();python.footer();python.left();python.contenet();cout << endl;cout << "cpp" << endl;Cpp cpp;cpp.header();cpp.footer();cpp.left();cpp.contenet();}int main(void){test();system("pause");return 0;}

• 使用继承的方法

• #include<iostream>#include<string>using namespace std;//公共页面class BasePage{public:void header(){cout << "首页、登录注册" << endl;}void footer(){cout << "帮助中心、交流合作" << endl;}void left(){cout << "java、python、c++" << endl;}};//普通实现页面//java页面class Java : public BasePage{public:void contenet(){cout << "java学科视频" << endl;}};class Python : public BasePage{public:void contenet(){cout << "python学科视频" << endl;}};class Cpp : public BasePage{public:void contenet(){cout << "c++学科视频" << endl;}};void test(){cout << "java" << endl;Java java;java.header();java.footer();java.left();java.contenet();cout << endl;cout << "python" << endl;Python python;python.header();python.footer();python.left();python.contenet();cout << endl;cout << "cpp" << endl;Cpp cpp;cpp.header();cpp.footer();cpp.left();cpp.contenet();}int main(void){test();system("pause");return 0;}

### 总结

* 继承的好处：减少重复代码

### 语法

* class 子类:继承方式 父类

### 概念

* 子类也称派生类
* 父类也称基类
* 派生类中的成员，包含量大部分
* 一类是从基类继承过来的，一类是自己增加的成员。
* 从基类继承过来的表现其共性，而新增加的成员体现其个性。

### 方式

* 公共继承

• 可以访问父类中的public和protected部分，且在子类中按原父类中的访问权限使用

* 保护继承

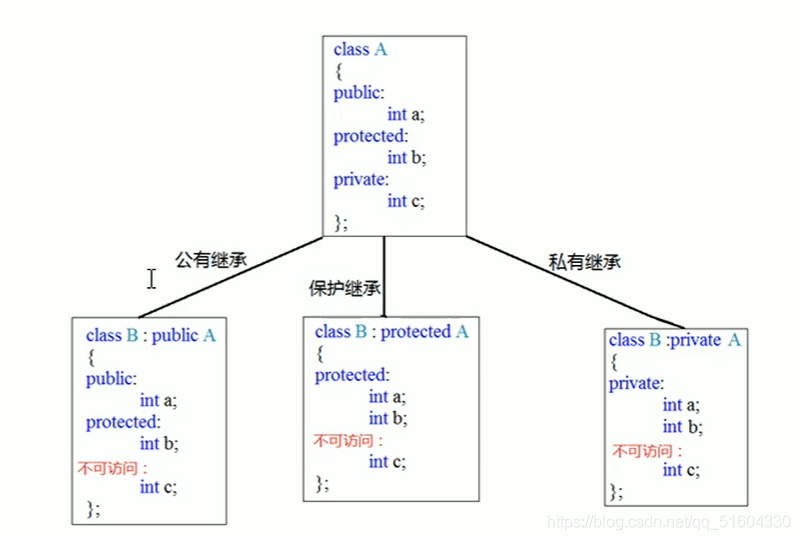
• 可以访问public和protected部分，且在子类中都以protected访问权限使用

* 私有继承

• 可以访问public和protected部分，且在子类中都以private访问权限使用

* 可视化图片

•



### 不同继承方式的案例

* #include<iostream>using namespace std;//公共继承class Base1{public:int m\_A;protected:int m\_B;private:int m\_C;};class Son1 :public Base1{public:void func(){m\_A = 10;//父类中的公共权限成员，到了子类中依然是公共权限m\_B = 20;//父类中的保护权限成员，到了子类中依然是保护权限//m\_C = 10;父类中的隐私权限成员，子类访问不到}};void test01(){Son1 son1;son1.m\_A = 100;//son1.m\_B = 100;保护权限的内容到了类外就无法访问了};//保护继承class Base2{public:int m\_A;protected:int m\_B;private:int m\_C;};class Son2 :protected Base2{void func(){m\_A = 100;//父类中公共权限的成员，因为是保护继承，到子类中变为保护权限m\_B = 100;//父类中保护权限的成员，保护继承后到了子类还是保护权限。//m\_C = 100;父类中的私有成员子类访问不到}};void test02(){Son2 son2;//保护权限类外访问不到，所以在son2中m\_A也访问不到了}//私有继承class Base3{public:int m\_A;protected:int m\_B;private:int m\_C;};class Son3:private Base3{void func(){m\_A = 100;//父类中公共成员，私有继承后，到了子类变为私有成员m\_B = 100;//父类中保护成员，私有继承后，到了子类变为私有成员//m\_C = 100;父类的私有权限成员仍然访问不到}};void test03(){Son3 son3;//私有成员类外访问不到}//验证Son3私有继承后成员是否变成了私有属性class GrandSon3 :public Son3{void func(){//访问不到父类的私有成员//到了Son3中m\_A,m\_B,m\_C全是私有成员，子类无法访问}};int main(void){system("pause");return 0;}

### 继承中的对象模型

* 问题

• 从父类继承过来的对象，哪些属于子类对象？

* 解答

• 父类中所有的非静态成员属性都会被子类继承下去。

• 父类中私有的成员属性是被编译器给隐藏了，因此访问不到，但是确实被继承下去了

### 继承中构造和析构的顺序

* 前言

• 子类继承父类后，当创建子类时，也会调用父类的构造函数。

* 问题

• 父类和子类的构造函数和析构顺序怎么样的呢？

* 解答

• 先构造父类，再构造子类

• 先析构子类，再析构父类

• 创建子类对象的同时也会创建一个父类对象。

* 案例

• #include<iostream>using namespace std;class Base{public:Base(){cout << "父类的构造函数" << endl;}~Base(){cout << "父类的析构函数" << endl;}};class Son:public Base {public:Son(){cout << "子类的构造函数" << endl;}~Son(){cout << "子类的析构函数" << endl;}};void test01(){Son son;}int main(void){test01();system("pause");return 0;}

* 总结

• 继承中先调用父类构造函数，再调用子类构造函数，析构顺序与构造相反。

### 继承同名成员处理方式

* 问题

• 当子类与父类出现同名的成员。如何通过子类对象，访问到子类或父类中同名的数据呢?

* 解答

• 访问子类同名成员，直接访问即可

• 访问父类同名成员，需要加作用域

* 案例

• #include<iostream>using namespace std;class Base{public:Base(){m\_A = 100;}void func(){cout << "父类同名成员函数调用" << endl;}void func(int a){cout << "父类同名重载成员函数调用" << endl;}int m\_A;};class Son:public Base {public:Son(){m\_A = 200;}void func(){cout << "子类同名成员函数调用" << endl;}int m\_A;};//同名成员属性处理方式void test01(){Son son;cout <<son.m\_A<< endl;//如果要通过子类对象访问到父类中的同名成员，需要加作用域。cout <<son.Base::m\_A<< endl;}//同名成员函数处理方式void test02(){Son son1; son1.func();//子son1.Base::func();//父//如果子类中出现和父类同名的成员函数//子类的同名成员会隐藏掉父类中所有同名成员函数//如果想要访问到父类中被隐藏的同名成员函数，需要加作用域son1.Base::func(10);}int main(void){test02();system("pause");return 0;}

* 总结

• 子类对象可以直接访问到子类中同名成员

• 子类对象加作用域可以访问到父类同名成员

• 当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类同名函数。

### 多继承语法

* C++允许一个类继承多个类
* 语法

• class 子类:继承方式 父类1，继承方式 父类2

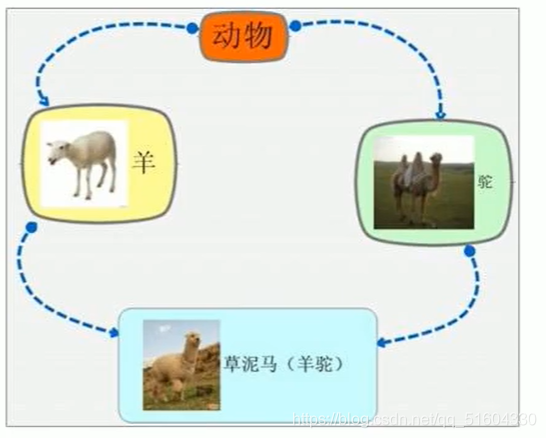
### 菱形继承

* 概念

• 两个派生类继承同一个基类，又有某个类同时继承这两个派生类，这种继承称为菱形继承，或者钻石继承。

* 典型案例

•



• 羊继承了动物的数据，驼同样继承了动物的数据，当草泥马使用数据时，就会产生二义性。

• 草泥马继承动物的数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

* 处理方法案例

• #include<iostream>using namespace std;class Animal{public:int m\_Age;};//利用虚继承可以解决菱形继承问题//在继承之前加上关键字virtual变为虚继承// Animal类称为虚基类//羊class Sheep:virtual public Animal{};//驼class Tuo:virtual public Animal{};//羊驼class SheepTuo :public Sheep,public Tuo{};void test01(){SheepTuo st;st.Sheep::m\_Age = 18;st.Tuo::m\_Age = 28;//当菱形继承，当两个父类拥有相同的数据，需要加作用域来区分cout << st.Sheep::m\_Age << endl;cout << st.Tuo::m\_Age << endl;cout << st.m\_Age << endl;//这份数据我们知道，只有一份就可以了，菱形继承导致数据有两份，资源浪费}int main(void){test01();system("pause");return 0;}

* 总结

• 菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义。

• 利用虚继承可以解决菱形继承问题——virtual

## 多态

### 分类

* 静态多态:函数重载和运算符重载属于静态多态，复用函数名
* 动态多态:派生类和虚函数实现运行时多态

### 区别

* 静态多态的函数地址早绑定 - 编译阶段确定函数地址
* 动态多态的函数地址晚绑定 - 运行阶段确定函数地址

### 案例

* #include<iostream>using namespace std;class Animal{public://加上virtual变成虚函数,实现地址晚绑定virtual void speak(){cout << "动物在说话"<< endl;}};class Cat :public Animal{public:void speak(){cout << "小猫在说话" << endl;}};class Dog : public Animal{public:void speak(){cout << "小狗在说话" << endl;}};//执行说话的函数//地址早绑定，在编译阶段就确定函数地址//如果想让猫说话，那么这个函数的地址就不能提前绑定，需要在运行阶段进行绑定//动态多条满足条件/\*1.有继承关系2.子类重写父类的虚函数\*///重写要求:函数返回值类型 函数名 参数列表 完全相同 //动态多态的使用/\*父类的指针或者引用 指向子类的对象//Animal &animal = cat;\*/void doSpeak(Animal &animal)//Animal &animal = cat;{animal.speak();}void test01(){Cat cat;doSpeak(cat);Dog dog;doSpeak(dog);}int main(void){test01();system("pause");return 0;}

### 总结

* 多态满足条件

• 有继承关关系

• 子类重写父类中的虚函数

* 多态的使用条件

• 父类指针或引用指向子类对象

* 重写:函数返回值类型 函数名 参数列表 完全一致称为重写

### 纯虚函数和抽象类

* 在多态中，通常父类汇中虚函数的实现是毫无意义的，主要都是调用子类重写的内容。因此可以将虚函数改为纯虚函数。
* 纯虚函数

• 定义

• 不能实例化对象，可以做指针和引用

• 作用

• 在基类中为其派生类保留一个函数的名字，以便派生类根据需要对他进行定义。如果在基类中没有保留函数名字，则无法实现多态性。

• 声明格式（语法）

• virtual 类型 函数名(参数列表）= 0

• 注意

• （1）纯虚函数没有函数体；

• （2）最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是虚函数”；

• （3）这是一个声明语句，最后有分号。

• 定义案例

• virtual void bark() = 0; //声明为纯虚函数

* 抽象类

• 定义

• 拥有纯虚函数的类是抽象类。

• 特点

• 无法实例化对象

• 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

* 案例

• #include<iostream>using namespace std;//纯虚函数和抽象类class Base{public://只要有一个纯虚函数,这个类称为抽象类//特点;无法实例化对象virtual void func() = 0;//注意:不要忘掉virtual!//抽象类的子类必须要重写父类中的纯虚函数，否则也属于抽象类};class Son :public Base{public:void func(){cout << "func函数调用" << endl;}};void test(){//Base b1; 抽象类无法实例化对象Son s1;//子类重写父类的虚函数，否则无法实例化对象Base\* abc = new Son;abc->func();}int main(void){test();system("pause");return 0;}

* 虚析构和纯虚析构

• 引入

• 多态使用的时候，如果子类中有属性开辟到堆区，那么父类指针在释放的时无法调用到子类的析构代码

• 解决方法

• 将父类中的析构函数改为虚析构或者纯虚析构

• 虚析构和纯析构共性

• 可以解决父类指针释放子类对象，

• 都需要有具体的含函数实现

• 虚析构和纯析构的区别

• 如果是纯虚析构，该类属于抽象类，无法实例化对象

• 虚析构语法

• virtual ~类名(){}

• 纯虚析构语法

• virtual ~类名() = 0;//声明

• 案例

• #include<iostream>#include<string>using namespace std;//虚析构和纯虚析构class Animal{public:Animal(){cout << "Animal的构造函数调用" << endl;}//利用虚析构可以解决父类指针释放对象时不干净的问题/\*virtual ~Animal(){cout << "Animal的析构函数调用" << endl;}\*///纯虚析构,需要声明也需要实现//有了纯虚析构之后，这个类也属于抽象类，无法实例化对象virtual ~Animal() = 0;//纯虚函数，不需要实现virtual void speak() = 0;};//纯虚析构函数Animal::~Animal(){cout << "Animal纯析构函数调用" << endl;}class Cat :public Animal{public:Cat(string name){m\_Name = new string(name);}virtual void speak(){cout << "Cat的构造函数调用" << endl;cout << \*m\_Name << "小猫在说话" << endl;}~Cat(){if (m\_Name != NULL){cout << "Cat的析构函数调用" << endl;delete m\_Name;m\_Name = NULL;}}string\* m\_Name;};void test01(){Animal\* animal = new Cat("Tom");animal->speak();/\*父类的指针在析构的时候，不会调用子类中的析构函数，导致子类如果有堆区属性，会出现内存的泄漏情况。解决:将父类的析构函数改为虚析构\*/delete animal;}int main(void){test01();system("pause");return 0;}

• 总结

• 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象问题

• 如果子类中没有堆区数据，可以不写为虚析构或纯虚析构

• 拥有纯虚析构函数的类也属于抽象类

# 写在前面，必看

## 这份总结是我根据自己的记忆进行整理的，可能有些内容的缺漏，但是重点难点肯定是不会缺省的。C++是一门实践课程，所以在每获取一个理论点后必须要结合实际去实践才能真正掌握，在这个思维导图中有很多代码，这些代码不是放在思维导图里面看的，而是把他们复制到自己的IDE中去运行看看到底为什么是这样子，这部分的代码都十分基础，理解起来不会有困难。如果C++还是搞不懂的建议上 b站看看黑马程序员的C++课程，深入浅出简单易懂，十分推荐。平时的题目如果遇到较长时间都打不出来的建议去CSDN看看别人的解法，不是一定要自己全部打出来，因为那样会比较浪费时间而且很低效，前面一直在学习和模仿后面才能打出自己的东西，而且CSDN的oj代码是很全的，所以多去学习，积极点，C++这门课不会有太大困难的