C++：封装、继承、多态

1、sizeof(数据类型/变量)，求占内存字节数

2、int arr[10];我们创建了一个int数组，任何时候记住在main函数或者任何函数中见到arr把他想成这个数组的首地址，只有在sizeof(arr)中不是让你求arr这个数组首地址占的内存字节数，二是求arr数组占的总内存字节数，这个时候我们可以通过sizeof(arr) / sizeof(arr[0])来求这个数组的元素个数，我们再拓展一下

```
int arr[2][3];
//那么arr表示这个二维数组的首地址。我们把第一行看作一个数组，那么arr[0]就是第一行数组的名字表示第一行数组的首地址，&arr[0][0],表示二维数组的首地址
```

3、常量指针和指针常量的区别：

const修饰指针---常量指针：指针的指向可以该，但是指针指向的内容不能改

```
int a = 10;
int b = 20;
const int * p = &a;//现在p就是个常量指针，他可以重新改为指向b
p = &b;//正确；
*p = 30;//错误；但是不能通过解引用修改指向内存的数值，这是不被允许的
const 叫常量 *叫指针；const ... * p 叫常量指针，表示*p不能改，就是内容不能改
```

const修饰常量 ---指针常量：指针的指向不可以改，但是指针指向的值可以改

```
int a = 10;
int b = 20;
int * const p = &a;//现在p就是个常量指针，他不可以重新改为指向b
p = &b;//错误；
*p = 30;//正确；能通过解引用修改指向内存的数值
const 叫常量 *叫指针；...  * const p 叫指针常量，表示p不能改，就是p中的内容不能改，就是p不能指向别人
```

const即修饰指针，又修饰常量：指针指向和指针指向的值都不可以被修改（易理解，不举例）

4、我们在main函数中调用其余函数，传的参数是指针和形参的区别，如果是指针，表示这个在函数中可以去改变这个指针指向的内容，但是如果是形参，相当于把实参复制了一份传给了函数，不能对实参进行修改

5、指针的注意事项：我们要把指针当作一种数据类型来看，就不迷糊了

```
int arr[4];
int *p = arr;//我们刚刚说过arr表示这个数组的首地址，让这个指针类型的数据中存储arr数组的首地址
++p;//指针指向向后移动该地址所代表数据的长度，到达第二个数据的地址，你不要想多，就是让p存储下一个数据的地址
p = p + 2;//就是做了两次p++;
p[3]//就是 p = p + 3;你看这个时候指针和arr数组首地址成一样用法了。
arr = arr + 3;//你看一个地址加三就是让这个地址往后跳三下，与上面是一样的意思。
```

6、冒泡排序：

```
void bubbleSort(int * p,int len){//p表示要排序数组的首地址，len表示这个数组中元素的个数
    for(int i = 0;i < len - 1;i++){
        for(int j = 0; j < len - 1 - i;j++){
            if(p[j] > p[j+1]){
                int tmp = p[j];
                p[j] = p[j + 1];
                p[j+1] = tmp;
            }
        }
    }
}
```

7、内存四区和new

C++中在程序运行前分为全局区和代码区两块内存
代码区：存放函数体的二进制代码，有操作系统进行管理。特点是只读和共享，比如一段二进制代码表示某个函数，那在程序运行中会被多次调用，这个时候是可以共享这块内存中的代码的，而且编译完成后，代码区的二进制文件就被固定下来了，就不能改了
全局区：存放全局变量、静态变量（static修饰）、常量（const修饰的全局常量和字符串常量）。程序在运行前会把所有代码中涉及到的“不变”的东西放在全局区，比如字符串常量，你用来输出的提示不管代码怎么运行都不会变，还有const修饰的全局变量
栈区：由编译器自动分配释放，存放函数的参数值，局部变量。比如main函数中的局部变量(实参)会在main函数运行结束释放，函数体中的形参会在函数运行结束后释放，以及一些运算过程中的临时变量和中间值，在函数执行过程中动态释放
堆区：由程序员分配和释放，若程序员不释放，程序结束时操作系统去回收。你疑惑这和你在main函数中int a = 10；有什么区别，你要明白这个a在main函数结束后会被释放，因为他本质上属于栈区的临时空间，他会被释放，我们想你在一个函数中的局部变量在函数运行结束返回main后还能查看他地址中的值吗？你不能，但是你如果用new在堆区创建他就可以。
不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程。
new运算符：

```
int * p = new int(10);//初始化为10
int * arr = new int[10];//new创建一个堆区的数组，返回这个堆区数组的首地址，用一个栈区的指针来接收就行。
```

我们注意new创建位于堆区的变量返回这个变量在堆区的地址，然后用一个栈区的指针p来存储这个堆区的地址
delete运算符：

```
delete p;
delete[] arr;
```

8、引用：（传参：值传递（复制）和地址传递）

引用：给变量起别名。

```
int a = 10;
int c = 30;
int &b = a;//这个时候存10的变量a别名也叫b
b = c;//不可以；引用必须初始化，初始化后就不可以更改，这个时候相当于改变了b的内容由10变为30
//引用的好处：函数传参时可以利用引用的技术让形参去修饰实参，可以简化指针修改实参

#include<iostream>
using namespace std;
```

```
//值传递,值传递不改变实参
void mySwap01(int a,int b){
    int tmp = a;
    a = b;
    b = tmp;
}
//地址传递改变实参
void mySwap02(int *a,int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
//引用传递改变实参，与地址传递一样
void mySwap03(int &a,int &b){
    int tmp = a;
    a = b;
    b = tmp;
}
//引用：给变量取别名int &b = a;给变量a起别名b
int main(){
    int a = 10;
    int b = 20;
    mySwap01(a,b);
    cout << "值传递不改变实参，a = " << a << ", b = "  << b <<
endl;
    mySwap02(&a,&b);
    cout << "地址传递改变实参，a = " << a << ", b = "  << b <<
endl;
    mySwap03(a,b);
    cout << "地址传递改变实参，a = " << a << ", b = "  << b <<
endl;
    return 0;
}
```

引用做函数的返回值：不要返回局部变量的引用 int &ref = test01();引用ref接收了test01函数返回的一个局部变量的引用。

引用的本质是一个指针常量：指针指向不可以修改，但是指针中的值可以修改，所以你返回一个局部变量（栈区）的指针常量是违法的的，因为栈区会被回收，你从函数中接收到一个局部变量的地址存储在一个指针常量中，结果转眼这个局部变量的内存被回收，而指针常量不可以修改指向，这样你就访问不

到这个值了。

```
//自动转换为 int* const ref = &a; 指针常量是指针指向不可改，也说明为什么引用不可更改
int& ref = a;
ref = 20; //内部发现ref是引用，自动帮我们转换为: *ref = 20;
```

常量引用：常量引用主要用来修饰形参，防止误操作，咱函数列表中可以加const来修饰形参，防止形参改变实参，你这样想，给指针常量加const，相当于const即修饰指针也修饰是常量，指向和指向的内容都不可以改变，这样的话你在定义一个函数的时候传入参数使用常量引用，你传参的时候传一个局部变量或者数组，你既不能改变这个数组的指向也不能改变这个数组中的值。但是又不是值传递，比较省内存

```
int &ref = a;等价于int * const ref = &a;
常量变量指的是：const int &ref = a;它等价于const int * const ref = &a;
定义：void showValue(const int &val){}
调用：int a = 10;showValue(a);
注意：还可以这样用---const int& ref = 10;是不是很奇怪，引用把这句翻译为int temp = 10;
const int& ref = temp;
```

c++推荐使用引用来代替指针，这样一来很方便

9、函数有默认参数，随查随用。

函数还有占位参数，但是目前阶段压根用不到占位参数；

10、函数重载：函数名可以相同，提高复用性。

重载的条件：同一个作用域下、函数名相同，函数参数的类型不同或者个数不同或者顺序不同

注意：函数的返回值不能作为函数重载的条件。就是函数的返回值的类型必须相同。

```
//函数重载的坑
//引用可以作为重载的条件
void func(int &a){

}
void func(const int &a){

}
void main(){
    int a = 10;
    func(a);//默认调用第一个
    func(10);//调用第二个，相当于const int &a = 10;这是合法的
}
```

函数重载遇到默认参数就不行了，会出现二义性，要避免这种情况，你写函数重载要避免默认参数

11、封装权限：

public: 公共权限--成员 类内可以访问，类外也可以访问----- 公共厕所

protected:保护权限--成员 类内可以访问，类外不可以访问，父子关系的情况下，子可以用父亲的成员

private:私有权限--成员 类内可以访问，类外不可以访问，只有自己可以用。儿子不可以用

12、struct和class的区别是struct的默认权限是公有权限，但是class是默认私有权限，且class可以有动作

13、构造函数和析构函数

```
//构造函数:类名(){};可以发生重载
//析构函数:~类名(){};不可以发生重载
class Person{
public:
    string name;
public:
    Person(){//无参构造函数
        cout << "Person无参构造函数的调用" << endl;
    }
    Person(string p_name){//有参构造函数
        name = p_name;
```

```
        cout << "Person有参构造函数的调用" << endl;
    }
    Person(const Person &P){//拷贝构造函数
        this.name = P.name;//将传进来的参数全部拷贝本对象身上
        cout << "Person拷贝构造函数的调用" << endl;
    }
    ~Person(){
        //你创建的类中有指向堆区的数据，把堆区开辟的空间释放干净，用
delete
        cout << "Person析构函数的调用" << endl;
    }
}

void main(){
    //构造函数的调用
    Person p;//无参构造函数调用
    person p_1(10);//有参构造函数
    Person p_2(p_1);//拷贝构造函数
    Person p_3 = Person(10);//有参构造，Person(10)叫做匿名对象，
匿名对象当前行结束后会直接回收
    Person p_4 = Person(p_3);//拷贝构造

}
```

C++中拷贝构造函数调用通常有三种情况：使用一个已经创建完毕的的对象来初始化一个新的对象；值传递的方式给函数传参；以值方式返回局部对象

你写一个类的时候C++默认情况下至少给你的类加三个函数：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）；默认拷贝构造函数，对属性进行值拷贝

浅拷贝：简单的赋值拷贝操作---我们默认拷贝构造函数都是浅拷贝，[27 类和对象-对象特性-深拷贝与浅拷贝*哔哩哔哩*bilibili]

深拷贝：在堆区重新申请空间，进行拷贝操作（主要用于类中有堆中的数据时进行深拷贝）

```
//初始化列表
class Person{
public:
    string name;
    int age;
    int score;
```

```
    int high;
public:
    //传统有参构造函数
    Person(string p_name,int p_age,int p_score,int p_high){
        name = p_name;
        age = p_age;
        score = p_score;
        high = p_high;
    }
    //初始化列表
    Person(string p_name,int p_age,int p_score,int
p_high):name(p_name),age(p_age),score(p_score),high(p_high);
}
```

当其他类对象作为本类的成员时，构造时先构造其他类，再构造本类，析构时先析构本类，再析构其他类，就这样想：创建一个人时先创建这个人的胳膊、腿等，再去创建这个人，析构时先杀人，再卸胳膊腿。

14、静态成员：

静态变量就是在普通变量前面加上static关键字，他和常量const的区别是，const修饰变量表示这个变量以后在程序运行中 不可以改变，static表示所有对象共用同一份数据，并非这个数据不可以改变，所以static只有在C++面向对象中有用，一边情况下他和一个普通变量一样

静态成员就是在类的成员变量和成员函数之前加上static关键字，称为静态成员，它分为：

- 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到全局区
    - 类内声明，类外初始化
- 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

```
class Person{
public:
    static int m_A;//静态成员变量
    static void func1(){//静态成员函数

    }
```

```
private:
    static int m_B;//静态成员变量也是有访问权限的，此时私有权限只能类内访问，类外不可访问
    static void func1(){//静态成员函数也是有访问权限的。

    }
}
int Person::m_A = 100;//类内声明，类外初始化
void main(){
    Person p1;
    Person p2;
    p1.m_A = 20;
    //此时p2中的m_A也变为20，因为p1和p2这两个对象共享同一份全局区的数据
    //静态成员变量的两种访问方式
    p1.m_A;
    Person::m_A;//不创建Person对象也能用
    //静态成员函数的两种访问方式
    p1.func1();
    Person::func1();
    return 0;
}
```

15、在C++中，类内的成员变量和函数分开存储，只有非静态成员变量才属于类的对象上，而静态成员变量会被存在全局区，它不属于类的对象上是指当你创建一个对象的时候，这个静态成员不占你创建的对象的内存的大小，你创建的对象处在栈区，而对象中的静态成员变量处在全局区，你的sizeof计算的是创建在栈区对象所占内存的大小，空对象（里面啥也不定义）占用内存空间为1，C++编译器会给每个空对象也分配一个字节空间，所以每个空对象也应该有一个独一无二的内存地址

16、this指针：**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，它的用途：

- 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，可以使用return *this；

33 类和对象-对象特性-this指针的用途*哔哩哔哩*bilibili

```
class Person{
public:
    int age;
    Person(int age){
        this->age = age;//this.age = age
    }
    Person returnMe(){
        return *this;//我们的返回是一个值返回，所以是本对象拷贝了一个和自己一摸一样的新的person返回去了。
    }
}
```

this指针的本质是 指针常量（Person * const this），指针的指向不可以修改但是指针指向的内容可以修改

17、const修饰成员函数

常函数：

- 成员函数后面加const后我们称这个函数为**常函数**
- 常函数不可以修改成员变量
- 成员变量声明时加关键字mutable后，在常函数中依然可以修改

常对象：

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

18、友元

在程序中，有些私有属性 也可以让类外特殊的一些函数或者类进行，就需要你将类外这些特殊的函数和类定义为该类的友元。

友元的目的是让一个函数或者类访问另一个类中的私有成员，其关键字为friend，友元的三种实现方式：

- 全局函数做友元
- 类做友元
- 成员函数做友元

```
class Building{
    friend void goodGay(Building * building);//全局函数goodGay是本类的友元函数
    friend class goodGayClass01;//类做友元
```

```
    friend void goodGayClass02::visit02();//goodGayClass02中的
成员函数visit02()做友元

public:
    Building(){
        m_SittingRoom = "客厅";
        m_BedRood = "卧室";
    }
public:
    string m_SittingRoom;
pricate:
    string m_BedRoom;
}
//全局函数做友元
void goodGay(Building * building){
    cout << "好基友全局函数访问你的卧室" << building->bedRoom <<
endl;//可以的成功
}
//类做友元
class goodGayClass01{
public:
    goodGayClass(){
        //创建一个建筑物的对象
        building = new Building;//回忆new返回什么？
    }
    ~goodGayClass(){
        delete building;
    }
    void visit(){
        cout << "好基友类正在访问：  " << building->bedRoom <<
endl;//好基友在访问私有成员
    }//参观函数，访问building指向的Building类中的卧室

    Building * building;
}
//goodGayClass02中的成员函数visit02()做友元
class goodGayClass02{
public:
    goodGayClass(){
        //创建一个建筑物的对象
        building = new Building;//回忆new返回什么？
    }
```

```
    ~goodGayClass(){
        delete building;
    }
    void visit01(){//不可以访问building->bedRoom
        cout << "好基友02类visit01正在访问： " << building->bedRoom << endl;//好基友在访问私有成员
    }//参观函数，访问building指向的Building类中的卧室----------出错
    void visit02(){//可以访问building->bedRoom
        cout << "好基友02类visit02正在访问： " << building->bedRoom << endl;//好基友在访问私有成员
    }//参观函数，访问building指向的Building类中的卧室----------正确
    Building * building;
}
```

19、**运算符重载**：对已有的运算符进行重新定义，赋予其另一种的功能，以适用不同的数据类型。

加号运算符重载：实现两个自定义数据类型相加的运算

```
//1、成员函数加号运算符重载
class Person{
public:
    int m_A;
    int m_B;
    Person(){
        this->m_A = 10;
        this->m_B = 20;
    }
    Person operator+(Person &p){//成员函数加号运算符重载
        Person temp;
        temp.m_A = this->m_A + p.m_A;
        temp.m_B = this->m_B + p.m_B;
        return temp;
    }
};
//2、全局函数加号重载
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;
}
int main(){
    Person p1;
    Person p2;
    Person p3 = p1+p2;//成员函数重载本质等价于Person p3 =
p1.operator+(p2)
    Person p4 = p1+p2;//全局函数重载本质等价于Person p4 =
operator+(p1,p2)
    Person p5 = p1+10;//全局函数重载本质等价于Person p5 =
operator+(p1,10)
    cout << p3.m_A << endl;
    return 0;
}
```

左移运算符：重载<< 配合友元可以输出自定义数据类型

[40 类和对象-C++运算符重载-左移运算符重载*哔哩哔哩*bilibili]

涉及到链式调用的概念(返回一个类)、cout<<本质、是否有输出类的私有属性的需求等。

```
class Person{
    friend ostream& operator<<(ostream& cout,Person &p);
private:
    int m_A;
    int m_B;
public:
    Person(){
        this->m_A = 10;
        this->m_B = 20;
    }
};
//成员函数实现不了<<，只能通过全局函数重载左移运算符
//void operator<<(ostream &cout,Person &p2){}//本质
operator<<(cout,p2) 简化cout << p2;但是返回值是空的，所以没法链式
调用，遇到cout << p1 << endl;就不管用了
ostream & operator<<(ostream &cout,Person &p2){
    Person temp;
    temp.m_A = p1.m_A + p2.m_A;
    temp.m_B = p1.m_B + p2.m_B;
    return temp;
}
```

```
int main(){
    Person p3;
    cout << p3 << endl;
```

递增运算符重载：通过重载递增运算符实现自己的整型数据

```
//自定义整型数据
class MyInteger{
friend ostream& operator<<(ostream& cout,MyInteger myint);
public:
    MyInteger(){
        m_Num = 0;
    }
private:
    int m_Num;
public:
    //成员函数重载前置++运算符：返回引用是为了可以一直对一个数据进行递增操作
    /*原理:为什么返回引用？我们对一个变量前置++，最后得到还是这个变量本身，所以我们要把这个对象自身返回去，如果我们返回值，那么返回的是这个变量的拷贝的副本，这样一来不符合要求，比如++(++myint)我们连做了两次++，第一次做++后如果返回对象的拷贝，第二次++就没办法让该对象再次++，之所以不返回指针是因为引用的本质就是指针常量，返回引用其实返回的就是这个变量本身，this是一个指针常量，他里面存的是对象本身的地址，我们对他解引用，得到是这个this指针指向的内容，也就是这个对象*/
    MyInteger & operator++(){
        //先进行++运算
        m_Num++;
        return *this;
    }
    //成员函数重载后置++运算符：int代表占位参数，告诉编译器用来区分前置和后置递增
    MyInteger operator++(int){
        //先记录当前值
        MyInteger temp = *this;//记录当前本身的值，然后让本身的值加一，但是返回的是以前的值的对象，达到先返回后++;
        m_Num++;
        return temp;//注意前置返回的是引用，后置是值返回

    }
}
```

```
//全局函数重载<<运算符
ostream& operator<<(ostream& cout,MyInteger myint){
    cout << myint.m_Num;
    return cout;
}
```

赋值运算符重载：C++默认至少给一个类增加四个函数：

- 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拷贝问题：

我们有一个类，里面有一个成员变量是指针，指向某一个堆区存储的值（new出来的某个变量），当我们用这个类实例化两个对象p1和p2，我们再用p1 = p2，把p2赋值给p1，这个时候C++默认提供的=是浅拷贝，让p2里面该指针存的内容（某个数的地址）原模原样的给p1的指针，这个时候我们发现这两个对象中的指针指向同一个堆区的数据，这个时候就不合适,比如下面这个析构函数，在程序结束时p1和p2都会析构一次，这样会对同一块内存释放两次，这样就不合适了。

```
class Person{
public:
    Person(int age){
        m_Age = new int(age);
    }
    ~Person(){
        if(m_Age != NULL){
            delete m_Age;
            m_Age = NULL;
        }
    }
    //成员函数重载赋值运算符：返回对象是本类是因为方便a = b = c这样的链式调用
    Person &operator=(Person &p){
        //编译器提供的=为浅拷贝，不合适
        //我们先要判断是否有属性在堆区，如果有先释放干净，然后再进行深拷贝
        if(m_Age != NULL){
            delete m_Age;
```

```
            m_Age = NULL;
        }
        m_Age = new int(*p.m_Age)
    }
    int * m_Age;
};
coid main(){
    Person p1(10);
    Person p2(20);
    p1 = p2;//使用了赋值运算符
    return 0;
}
```

关系运算符重载：重载关系运算符，可以让两个系定义类型对象进行对比操作

```
class Person{
public:
    Person(string name,int age){
        this->m_Name = name;
        this->m_Age = age;
    }
    int m_Age;
    string m_Name;
    //重载==运算符
    bool operator==(Person &p){
        if(this->n_Name == p.m_Name && this->n_Age ==
p.m_Name){
            return true;
        }
        else{
            return false;
        }
    }
    //重载==运算符
    bool operator!=(Person &p){
        if(this->n_Name == p.m_Name && this->n_Age ==
p.m_Name){
            return false;
        }
        else{
            return true;
        }
```

```
    }
}
void main(){
    Person p1("TOM",12);
    Person p2("TOM",12);
    if(p1 == p2){
        cout << "关系运算符==重载" << endl;
    }
    else{
        cout << "关系运算符!="重载 << endl;
    }
    return 0;
}
```

函数调用运算符重载：（知道即可，用不到）

- 函数调用运算符{}也可以重载
- 由于重载后使用方法非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

```
class MyPrint{
    void operator()(string test){
        cout << test << endl;
    }
}
void main(){
    MyPrint myprint;
    myprint("我是你爹");//由于使用起来非常像函数时调用，所以叫仿函数
    return 0;
}
```

20、继承{class　子类：继承方式　父类}　注意：子类也成为派生类，父类也称为基类

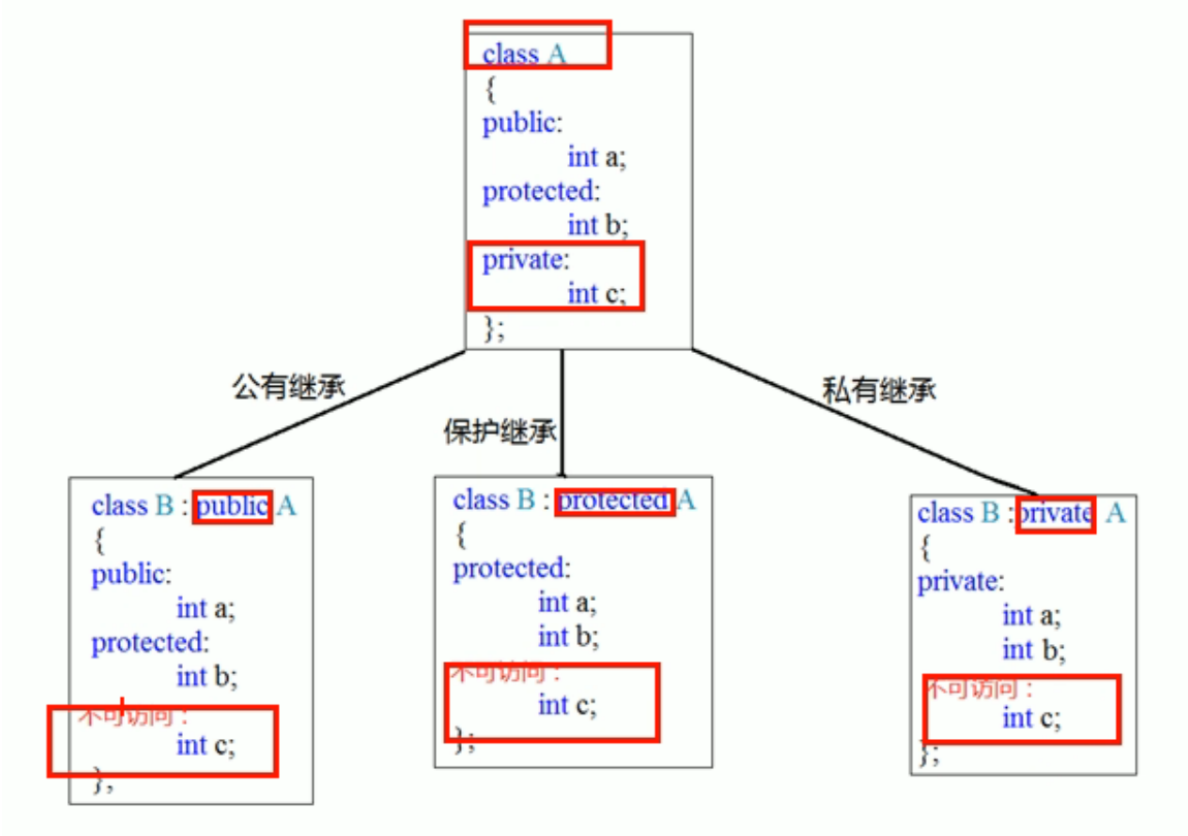
子类中的成员有两大类：一类是从父类继承过来的，一类是自己的成员，从父类继承过来的表现其共性，而新增的成员体现其个性。

父类的成员有三类：**私有成员**，**保护成员**和**公有成员**。

继承方式：

- 公共继承
- 保护继承

- 私有继承



注意：

- 无论是哪种继承方式，父类的私有成员子类都不可以访问
- 父类所有非静态成员属性都会被子类继承到，包括父类的私有成员，只是子类无法访问而已，并不代表子类不保存父类的私有成员，这些父类的私有成员被编译器给隐藏了。

子类继承父类后，当创建子类对象时，也会调用父类的构造函数去给子类对象中拥有的父类成员初始化，而且该调用是自动调用，但问题是父类和子类的构造函数和析构函数是谁先谁后？

- 答：父类的构造函数---->子类的构造函数---->子类的析构函数----->父类的析构函数

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父类中同名的数据？

- 访问子类同名对象，直接访问即可
- 访问父类同名对象，需要加作用域
- 当子类与父类拥有同名成员函数，子类会隐藏掉父类中同名成员函数，加作用域可以访问到父类中的同名成员函数

```
//成员包括成员函数和成员变量，我们调用方法是一样的
//父类
class Base{
public:
    Base(){
        m_A = 100;
    }
    int m_A;
};
//子类：子类中的m_A与父类中的m_A同名
class Son :public Base{
    public:
    Son(){
        m_A = 100;
    }
    int m_A;
}
int main(){
    Son s;
    cout << "Son下的m_A = " << s.m_A << endl;
    cout << "Base下的m_A = " << s.Base::m_A << endl;//访问父类中的同名成员要加作用域，同名成员函数也一样要加作用域，同名成员重载函数也一样要加作用域
    return 0;
}
```

问题：继承中同名静态成员在子类对象上如何进行访问?

- 答：静态成员和非静态成员出现同名时，处理方式一致，但注意这两个静态成员决不是同一个静态成员

```
class Base{
public:
    static int m_A;
}
int Base::m_A = 100;
class Son{
public:
    static int m_A;
}
int Son::m_A = 200;
int main(){
```

```
    //静态成员变量有两种访问方式
    //第一种
    Son s;
    s.m_A;//儿子的m_A
    s.Base::m_A//爹的m_A
    //第二种
    Son::m_A;
    Son::Base::m_A;//Son::表示通过类名方式访问，Base::代表访问父类作用域下
}
//上述代码我们写了static修饰的父子同名变量访问方式，它与static修饰的父子同名函数访问方式一致
//同名静态成员处理方式和非静态处理方式一样，只不过有两种访问方式（通过对象和通过类名）
```

多继承：C++与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.....

注意：多继承可能会引发父类中有同名成员出现，需要加作用域区分，**C++实际开发中不建议多继承**

菱形问题以及其解决方案：52 类和对象-继承-菱形继承问题以及解决方法*哔哩哔哩*bilibili使用虚继承来解决这个问题，课中有。

21、多态：顾名思义就是多种形态

多态分为两类：

- 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

静态多态和动态多态的区别：

- 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

```
class Animal{
public:
    virtual void speak(){
        cout << "动物在说话" << endl;
    }
};
```

```
//猫类
class Cat :public Animal{
public:
    void speak(){
        cout << "小猫在说话" << endl;
    }
};
//执行说话的函数，这个参数传递是父类的引用在指向一个子类的对象而无需类型强制转换。
//地址早绑定，在编译阶段就确定了函数的地址
void doSpeak(Animal &animal){//Animal & animal = cat
    animal.speak();
}
void test01(){
    Cat cat;
    doSpeak(cat);
}
```

运行text01，结果是“动物在说话”-----原因：void doSpeak(Animal &animal)是地址早绑定，在编译阶段就确定了函数的地址，animal.speak()中animal是Animal类的地址，那么doSpeak函数在编译阶段让animal.speak()的speak()提前绑定了Animal类的speak函数的地址，如果想执行让猫说话，那么这个函数地址就不能提前绑定，需要在运行阶段进行绑定，也就是地址晚绑定

怎么实现地址晚绑定？答：我们在Animal类中把speak函数前面加个virtual关键字，将其变为虚函数，就可以实现地址晚绑定了-----这就是通过虚函数实现运行时多态的方法

动态多态的满足条件：（**虚函数实现运行时多态**）

- 有继承关系
- 子类要重写父类的虚函数（重写和重载不同，重写是函数返回值、函数名、参数列表完全相同）
- 子类重写的父类虚函数前面可加也可不加virtual

多态的原理剖析：

```
class Animal{
public:
    virtual void speak(){
        cout << "动物在说话" << endl;
    }
```

```
};
//猫类
class Cat :public Animal{
public:
    coid speak(){
        cout << "小猫在说话" << endl;
    }
};
//执行说话的函数，这个参数传递是父类的引用在指向一个子类的对象而无需类型强制转换。
//地址早绑定，在编译阶段就确定了函数的地址
void doSpeak(Animal &animal){//Animal & animal = cat
    animal.speak();
}
void test01(){
    Cat cat;
    doSpeak(cat);
}
void test02(){
    cout << "sizeof Animal" << sizeof(Animal) << endl;//空类占1个字节
}
int main(){
    test01();
    test02();//运行结果为1
    return 0;
}
```

我们发现上述代码给Animal中的speak再加上virtual之前test02()执行结果为1，加上virtual之后test02()执行结果为4，我们之前说过一个空的类就是1个字节，用来占位

原理剖析：[54 类和对象-多态-多态的原理剖析*哔哩哔哩*bilibili]

```
class Person{
public:
    int m_A;
    int m_B;
    void func(){
        cout << "傻逼" << endl;
    }
}
```

一个类里面有成员变量和成员函数，我们用该类创建了一个对象，查看这个对象所占内存大小的时候我们发现只有该对象成员变量所占的大小，没有成员函数所占字节数

因为 **成员函数不属于"对象本身"**，它们属于"类"，对象里只存"数据"，不存"代码"。

更具体点：

1) 对象的内存布局 = 成员变量（+ 可能的额外隐藏指针）

你 `sizeof(对象)` 看到的，是这块对象内存里实际要存的东西，主要是：

- 各种成员变量
- 对齐填充（padding）
- **如果有虚函数**，还会多一个隐藏的 **vptr（虚表指针）**（通常 8 字节，64 位环境）

所以你看到的基本就是"数据大小"。

2) 成员函数的代码放在哪?

成员函数编译后是一段机器码，放在程序的 **代码段（text segment）**，全局只有一份。

所有对象调用同一个成员函数，其实是：

- 编译器把 `obj.f()` 变成类似 `f(&obj)` 这样的调用（隐式传 `this`）
- 代码不需要在每个对象里复制一份，否则创建 1000 个对象就要复制 1000 份函数代码，太离谱

这就很好解释为什么上述代码给Animal中的speak再加上virtual之前test02()执行结果为1，因为编译器在编译的时候就已经把animal.speak()绑定到了Animal类的代码区了，你再去看视频很快就能理解

多态：父类**指针**或**引用**指向子类对象，通过父类调用被子类重写的成员函数。

纯虚函数和抽象类： 56 类和对象-多态-纯虚函数和抽象类_哔哩哔哩_bilibili

在多态中，通常父类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**，纯虚函数的写法：`virtual 返回值类型 函数名（参数列表）= 0;`当类中只要有纯虚函数，这个类也称为**抽象类**

抽象类的特点：

- 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
- 虚函数就是给函数前面virtual，纯虚函数是：`virtual 返回值类型 函数名（参数列表）= 0;`

虚析构和准虚析构：[58 类和对象-多态-虚析构和纯虚析构_哔哩哔哩_bilibili](#)

问题：多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码

解决方案：将父类中的析构函数改为**虚析构**或者**纯虚析构**(其本质就是析构函数的多态)

虚析构和**纯虚析构**共性：

- 可以解决父类指针释放子类对象
- 都需要具体的函数实现

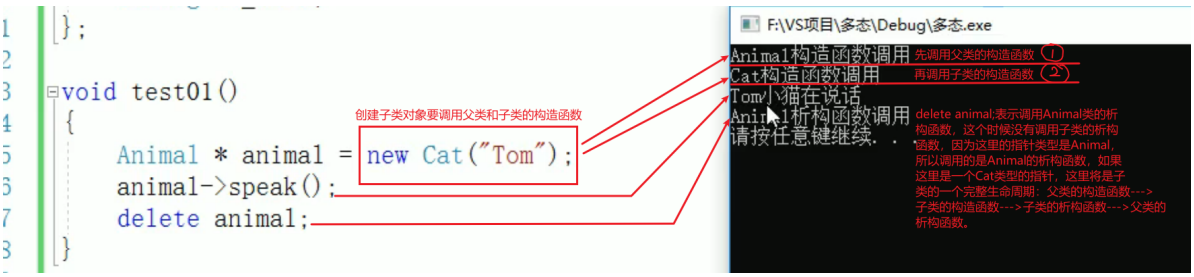
虚析构和**纯虚析构**的区别：

- 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;`需要外部定义`类名::~类名(){}`

在这里我们要**搞清楚**这么一件事情：

子类继承父类后，当创建子类对象时，也会调用父类的构造函数去给子类对象中拥有的父类成员初始化，而且该调用是自动调用，但问题是父类和子类的构造函数和析构函数是谁先谁后？

- 答：父类的构造函数---->子类的构造函数---->子类的析构函数----->父类的析构函数



虚析构就是我们让父类的析构函数变为虚析构函数，这样delete animal；就是先执行子类的析构函数，然后执行父类的析构函数，这样一来上述test01的代码执行顺序为：父类的构造函数--->子类的构造函数--->子类的析构函数--->父类的析构函数。非常符合我们以前的使用习惯，如下图：

```
Animal()
{
    cout << "Animal构造函数调用" << endl;
}

virtual ~Animal()
{
    cout << "Animal析构函数调用" << endl;
}
//纯虚函数
virtual void speak() = 0;
```

```
F:\VS项目\多态\Debug\多态.exe
Animal构造函数调用
Cat构造函数调用
Tom小猫在说话
Cat析构函数调用
Animal析构函数调用
请按任意键继续. . .
```

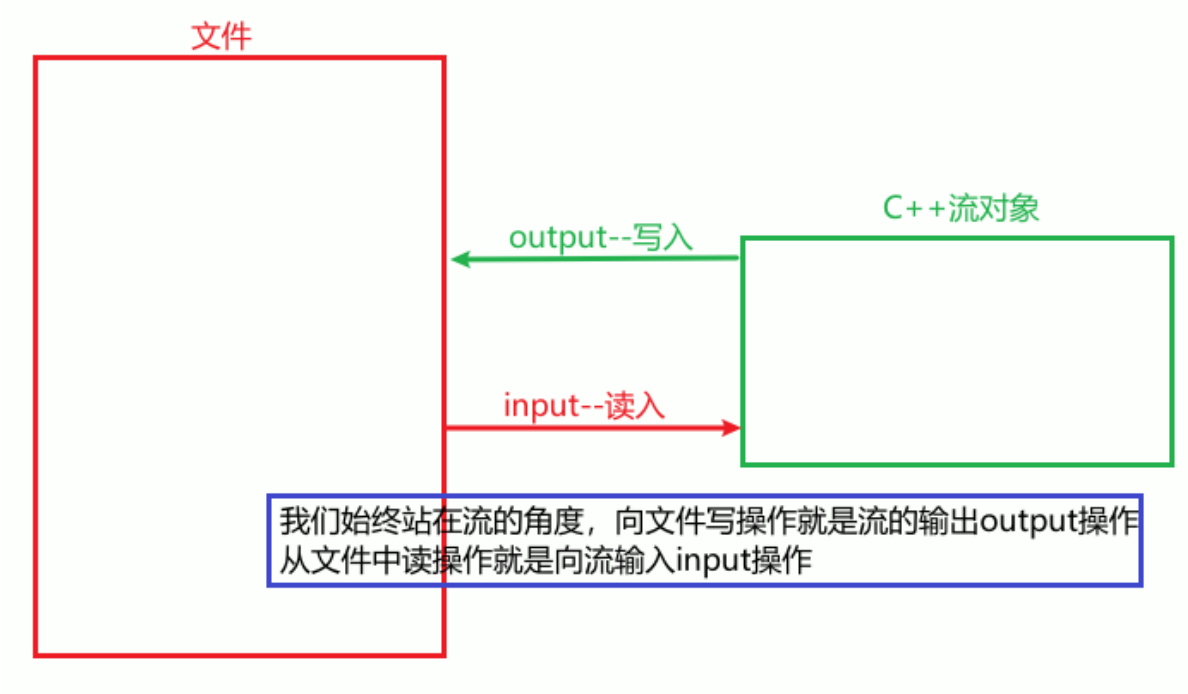
纯虚析构：`virtual ~Animal() = 0;`外部定义：`类名::~类名(){}`

总结：

- 虚析构和纯虚析构就是用来解决通过父类指针释放子类对象
- 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

22、文件操作

对流的重要理解，我们要站在流的角度来理解**读和写**



程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放，通过**文件将数据持久化**，C++对文件的操作要包括头文件，文件分为两种类型：

- **文本文件**：文件以**文本ASCII码**形式存储在计算机中
- **二进制文件**：文件以**文本的二进制**形式存储在计算机中，用户很难读懂

操作文件的三大类：

- ofstream：写操作
- ifstream：读操作
- fstream:读写操作

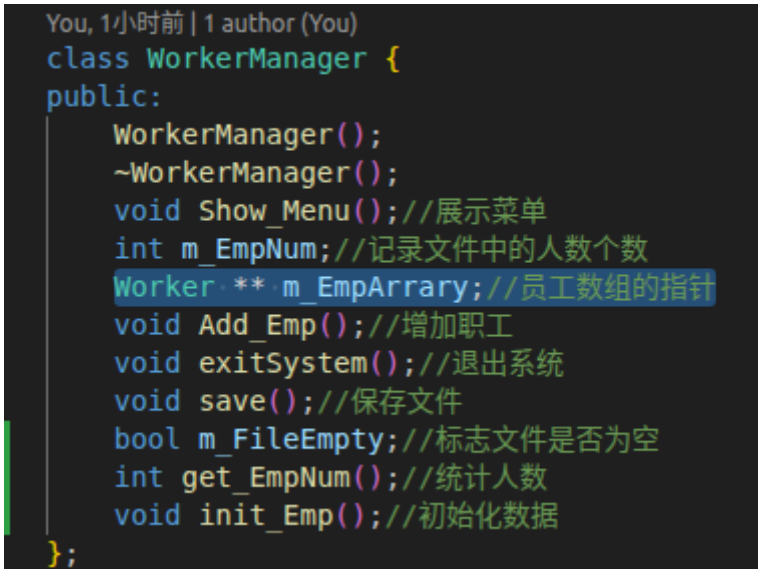
```
/*
写文件：
1、包含头文件：#include<fstream>
2、创建流对象 ofstream ofs;
3、打开文件 ofs.open("文件路径",打开方式);
4、写数据 ofs << "写入的数据";
5、关闭文件 ofs.close();
*/


/*
读文件：
1、包含头文件：#include<fstream>
2、创建流对象 ifstream ifs;
3、打开文件 ifs.open("文件路径",打开方式);判断文件是否打开成功
（bool） ifs.is_open()
4、读数据（4种）
5、关闭文件 ifs.close();
*/
```

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置：文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

```
//第一种
char buf[1024] = {0};
while(isf >> buf){
```

```
    cout << buf << endl;
}
//第二种
char buf[1024] = {0};
while(ifs.getline(buf,sizeof(buf))){
    cout << buf << endl;
}
//第三种推荐
string buf;
while(getline(ifs,buf)){
    cout << buf << enldl;
}
//第四种不推荐
char c;
while((c.ifs.get()) != EOF){
    cout << c;
}
```

23、二级指针



`Worker ** m_EmpArrary;` 这句可以用一句话理解：

> **它是"指向 `Worker *` 的指针"**，也就是**二级指针**，通常用来表示"一个动态的指针数组"（数组里每个元素都是 `worker*`，指向不同的员工对象）。

你可以把它想成：
m_EmpArrary → 一张"员工地址表"，表里每一格存着某个员工对象的地址。

因为你要存的是"很多个员工"，而且每个员工可能是不同类型（比如 Employee / Manager / Boss），通常会用基类指针存：

- 单个员工：`Worker* p;` （指向一个 Worker 或其子类对象）
- 多个员工（数组里放很多个 `Worker*`）：`Worker** arr;`

它本质上对应这种结构：

```
Worker* arr[100];   // 固定大小的“指针数组”
```

但你现在要动态大小（人数会变），就用动态分配：

```
Worker** m_EmpArrary = new Worker*[m_EmpNum];
```

这时候：

- `m_EmpArrary` 的类型就是 `Worker**`
- `m_EmpArrary[i]` 的类型就是 `Worker*`
- `*(m_EmpArrary[i])` 才是那个员工对象（Worker）

假设有 3 个员工：

```
m_EmpArrary  ---->  [ 0xAAA ] [ 0xBBB ] [ 0xCCC ]
                      |         |         |
                      v         v         v
                   Employee   Manager    Boss    (这些对象在别处
new 出来)
```

- `m_EmpArrary` 指向一段连续内存（数组）
- 数组里每个元素都是一个 `Worker*`
- 每个 `Worker*` 指向一个真实员工对象（可能是不同派生类）

因为你分配了两种东西：

1. 每个员工对象：`new Employee/Manager/Boss`
2. 指针数组本体：`new Worker*[n]`

所以析构/清空要这样：

```
for (int i = 0; i < m_EmpNum; ++i) {
    delete m_EmpArrary[i];   // 删除每个员工对象
}
delete[] m_EmpArrary;        // 删除指针数组
m_EmpArrary = nullptr;
```