C++初级学习教程

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include<iostream>

using namespace  std;

struct Teacher
{
	char name[10];
	int age;
};
	const 修饰的是指针指向的内存空间，因此成员变量表不能修改
int operateTeacher1(const Teacher*PT)
{
	//PT->age = 1;
	return 0;
}
	//指针变量本身不能被修改，但是可以修改成员变量
int operateTeacher2( Teacher*const PT)
{
	PT->age = 1;
	//PT= NULL;
	return 0;
}
	//都不能被修改
int operateTeacher2( const Teacher*const PT)
{
	//PT->age = 1;
	//PT= NULL;
	return 0;
}
/*****************************引用************************************/
/*
	函数返回值是引用：
	c++引用时的难点：
```

当函数返回值为引用时，若返回的是变量，不能成为其它引用的初始值，不能作为左值使用

若返回静态变量或者全局变量，可以成为其它引用的初始值，可以作为左值和右值使用

指针的引用，常量的引用　引用是变量的别名

const Teacher &Myt 常引用，让实参变量 拥有只读属性，相当于

const Teacher *const Myt;

Myt.age = 50; //出错

/**************************C++ 对 C 的拓展**************************/

（1）inline 函数

（2）默认参数

（3）函数的占位参数

（4）函数重载

一、内联函数

C++推荐使用内敛函数代替宏代码片段

inline MyFun(a,b) ((a) < (b) ? (a) :(b))

C++编译器直接将函数体插入函数调用的地方

内联函数是有编译器进行处理，宏代码片段由预处理器处理，进行简单的文本替换，没有任何编译过程

C++内联编译的限制：

(1)不能存在任何性质的循环语句

(2)不能存在过多的条件判断语句

(3)函数体不能过于庞大

(4)不能对函数进行取地址操作

(5)函数内联声明必须在调用语句之前

(6)编译器对于内联函数的限制并不是绝对的，内联函数相对普通函数的优势只是省去了函数调用时的压栈、跳转和返回的开销

因此，当函数体的执行开销大于压栈、跳转和返回所用的开销的时候，那么内联函数将会毫无意义。

(7)内联函数只是一种啊请求，编译器不一定允许这种请求

二、默认参数

在函数的申明的时候，默认参数必须放在最右边：

MyFun(int M,int m = 10,int N = 10);

三、占位参数

（1）占位参数只有参数类型的声明，没有参数名

（2）一般情况下，在函数体内无法使用占位参数

（3）可以把占位参数和默认参数结果起来使用，意义：用于程序拓展和 C 的兼容

四、函数重载

同一个函数名定义不同的函数

当函数名和不同的参数搭配时候函数的含义不相同

函数重载至少满足下列的一个要求：

1)参数个数不同、2）参数类型不同、3）参数顺序不同

函数的返回值不是函数重载的判断标准

重载函数是由函数名称和参数列表构成

两个难点：

（1）重载函数和默认函数参数混搭

MyFun(int a,int b,int c = 0) MyFun(int a,int b)

MyFun(1,2) //C++编译器无法确定调用哪个函数，编译不通过，报错

（2）重载函数和函数指针

*/

/****************************** C++ 对 C 的拓展*****************************/

/*

C++面向对象编程思想：工具：面向抽象类编程（面向接口、软件分层）

封装:（1）对属性方法封装 （2）属性方法的访问控制

*/

/***/

```
class MyCircle
{
public:
	double m_r; //圆的半径
	double m_s; //圆的面积

	double GetR(){return m_r;}
	double GetArea(){return 3.14 * m_r *m_r;}
	void SetR(double R)
	{
		m_r = R;
	}
};
void PrintCircle(MyCircle *myCircle)
{
	double R = myCircle->GetR();
	double Area = myCircle->GetArea();
```

```
	cout << "R = "<<R<< "   "<<"Area = "<< Area <<endl;

}
void PrintCircles(MyCircle &myCircle)
{
	double R = myCircle.GetR();
	double Area = myCircle.GetArea();
	cout << "R = "<<R<< "   "<<"Area = "<< Area <<endl;

}
void PrintCircl(MyCircle myCircle)
{
	double R = myCircle.GetR();
	double Area = myCircle.GetArea();
	cout << "R = "<<R<< "   "<<"Area = "<< Area <<endl;

}
void main_01()
{
	MyCircle c1,c2;
	c1.SetR(10),c2.SetR(25);
	PrintCircles(c1);
	PrintCircle(&c2);

	system("pause");

}
	//设计立方体类，计算立方体的体积，判断立方体是否相同
	//成员函数和全局函数的区别
class MyCube
{
private:
	int m_a,m_b,m_c;
public:
	int GetA(){return m_a;}
	int GetB(){return m_b;}
	int GetC(){return m_c;}
public:
```

```
    int GetVolume(){return m_a * m_b *m_c;}

    void SetElement(int a = 0,int b = 0,int c = 0)
    {
        m_a = a, m_b = b, m_c = c;
    }
    //判断两个立方体的长宽高是否相同 使用的是面向对象的过程
    int JudgeCube(MyCube &Cube)
    {
        if(m_a == Cube.GetA() && m_b == Cube.GetB() && m_c == Cube.GetC())
            return 1; //园内
        else
            return 0;//圆外
    }

};
    //全局函数，面向过程的
int JudgeCube(MyCube &iCube, MyCube &mCube)
{
    if(iCube.GetA() == mCube.GetA() && iCube.GetB() == mCube.GetB() &&
iCube.GetC() == mCube.GetC())
        return 1;
    else
        return 0;
}
void main_02()
{
    MyCube Cube1,Cube2;
    Cube1.SetElement(1,2,3);
    Cube2.SetElement(1,5,3);
    int flag = Cube1.JudgeCube(Cube2);
    if(flag == 1)
        cout << "Same" << endl;
    else
        cout << "difference"<< endl;
    system("pause");
}
    //设计一个圆形类和一个点类，计算点和圆的位置关系
```

```
class MyPoint
{
private:
	int x;
	int y;
public:
	int GetX(){return x;}
	int GetY(){return y;}
	void SetPointOrdinate(int _x,int _y)
	{
		x = _x;
		y = _y;
	}
};

class MyCircles
{
private:
	int m_r;
	int m_x;
	int m_y;
public:
	void SetCircleElements(int _r, int _x, int _y)
	{
		m_r = _r, m_x = _x, m_y = _y;
	}
	int JudgeTypt(MyPoint &MyP)
	{
		int x_value = MyP.GetX();
		int y_value = MyP.GetY();
		float dis = sqrt((float)pow((float)(x_value - m_x),2) + pow((float)(y_value -
m_y),2));
		if(dis < m_r)
			return 1;
		else
			return 0;
	}
```

```
};
void main_03()
{
	MyPoint pt;
	pt.SetPointOrdinate(7,3);
	MyCircles Circle;
	Circle.SetCircleElements(5,2,2);
	int flag = Circle.JudgeTypt(pt);
	if(flag == 1)
		cout << "点在园内"<<endl;
	else
		cout << "点在圆外"<<endl;

	system("pause");
}
/*************************************************************************/
/*
	类和对象、对象的构造和析构、静态成员变量和成员函数、面向对象模型初探
	编译器对属性和方法的处理机制、 this 指针、友元函数、友元类
	一、对象的构造和析构
	创建一个对象的时候，需要做某些初始化的工作，类的数据成员是不能在类的申明的时候初始化
	为了解决该问题，C++编辑器提供了构造函数来处理对象的初始化，构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用具的调用，而是在建立对象的时候自动执行
	构造函数在定位的时候可有有参数；
	没有任何返回类型的声明
	构造函数的调用：一般情况下 C++自动调用，手动调用
	析构函数没有参数，也没有任何类型的声明
	析构函数在对象销毁的时候调用
	析构函数是由 C++自动调用
	构造函数分为三种：无参构造函数、有参构造函数、复制类型的构造函数
	（1）拷贝构造函数是采用一种对象初始化另一个对象
	必要的时候，需要程序员手动的编写拷贝构造函数
	一个对象的初始化列表
*/
/*************************************************************************/
```

```
    //拷贝构造函数的四种调用实际

class Test
{
public:
    Test() {    };//无参的构造函数
    Test(int a,int b) :m_a(a),m_b(b){cout << m_a <<"  "<< m_b<<endl;   }//有参构造函数
    Test(const Test& Obj)//赋值构造函数
    {
        m_a = Obj.m_a + 100;
        m_b = Obj.m_b + 100;
    }
    void PrintE()
    {
        cout << "m_a = " << m_a << "m_b = "<<m_b << endl;
    }
private:
    int m_a;
    int m_b;

};
void main_04()
{
    Test t1; //无参的构造函数
    Test t2(2,3);//有参构造函数调用
    Test t3 = t2;//用 t2 初始化 t3    (1)种
    t3.PrintE();
    Test t0(5,3);
    t0 = t2; //用 t2 对 t0 赋值，赋值造作不会调用构造函数
    Test T4(t2); //用 t2 初始化 t4     (2)种
    system("pause");

}
class Location
{
private:
    int x;
    int y;
```

```
public:
	Location(int _x,int _y):x(_x),y(_y)
	{
		cout << "x="<<x<<"  y="<<y<<endl;
	}
	Location(const Location& obj)
	{
		x = obj.x + 20;
		y = obj.y + 30;
		cout << "x="<<x<<"  y="<<y<<endl;
	}
	~Location()
	{
		cout << "x="<<x<<"  y="<<y<<endl;
	}
	int CalX(){return x;}
	int CalY(){return y;}

};
void MyFun(Location Loc)
{
	cout << Loc.CalX();
}
void main_05()
{
	Location L1(2,5);
	Location L2(L1);
	MyFun(L1);//实参给形参赋值的时候，C++编译器会自动的调用构造函数
	cout<<"HELLO!"<<endl;
	system("pause");

}

	//浅拷贝和深拷贝

class Name
{
private:
```

```
    int lenth;
    char *p;
public:
    Name(const char *MyChar)
    {
        lenth = strlen(MyChar);
        p = (char *)malloc(lenth+1);
        strcpy(p,MyChar);
    }
    //解决方案，手动编写拷贝构造函数，采用深拷贝
    Name(const Name &MyName)
    {
        lenth = MyName.lenth;
        p = (char *)malloc(lenth +1);
        strcpy(p,MyName.p);
    }
    ~Name()
    {
        if(p != NULL)
        {
            free(p);
            p = NULL;
            lenth = 0;
        }
    }
};
void main_06()
{
    Name N("ABC");

    Name M = N;  //调用的是拷贝构造函数
    //采用 C++编译器自带的赋值构造函数进行初始化（属于浅拷贝）M,N 是指向同一个内存区间的两个对象
    //在析构的时候出错。在析构过程中先对 M 进行析构，释放内存空间，对 N 释放的时候会出现错误
    //针对以上问题，则需要进行深拷贝，为 M 也开辟新的内存空间，需要程序员构建新的拷贝构造函数
    Name O("abc");
```

O = N; //赋值操作是将对象 N 的属性值拷贝给对象 O,属于浅拷贝

}

/***/

/*

多个对象之间的相互操作：多个对象初始化列表

构造函数的初始化列表解决了在一个类中组合了另一个类的对象，该类设计了有参构造函数

先执行被组合对象的构造函数，然后执行调用对象构造函数，析构函数相反

初始化列表用来对 Const 变量进行初始化

static 类型提供了类之间的共享机制。所有的对象共享该成员

*/

/***/

/*

C++面向对象初探

C++模型可以概括为下列两个部分：

(1) 语言中直接支持面向对象过程程序设计的部分，主要包括构造函数、析构函数、虚函数、继承（单继承、多继承、虚继承、）、多态等。

(2) 对于各种支持的底层实现机制：在 C++中通过抽象数据类型，在类中定义数据和函数，来实现数据和函数的直接绑定。

概括来说，C++中有两种成员数据，static,nonstatic;三种成员函数：static,nonstatic，virtual;

C++编译器是如何管理类、对象、类和对象之间的关系；具体来说，具体对象调用类中的方法，C++编译器是如何区分是个具体的类，调用的是哪个方法

C++编译器对属性和方法的处理机制

(1)C++中的成员变量和成员函数是相互分开存储的

成员变量：

普通成员变量：存储在对象中，和 struct 有相同的内存布局和对其方式；

静态成员变量: 存储在全局数据区

成员函数：存储在代码段中。

那么问题来了：很多对象公用一个代码，代码如何区分具体的对象是哪个？使用隐含的 This 指针

总结：

C++类对象中的成员变量和成员函数是分开存储的；

C++中的普通成员函数都隐式包含一个指向当前对象的 This 指针

静态成员变量、函数属于类

静态成员函数与普通成员函数的区别

静态成员函数不包含指向具体对象的指针

普通成员函数包含一个指向具体对象的指针

```
Class Test
 {
 private:
            int a ,int b;
 public:
        void const operateVar(int a ,int b);
        const void operateVar(int a ,int b) ;
        void  operateVar(int a ,int b) const  => void operateVar（const Test *this, int a,int b）;
 }
```

const 修饰的是 This 指针指向的内存空间，因此无法对 This 指针的所指向的空间变量进行修改,让 a,b 只具有只读属性

```
*/
/*************************************************************************/
```

//全局函数 PK 成员函数

```
class CTest
{
public:
    int a;
    int b;
public:
    CTest(int a ,int b)
    {
        this->a = a;
        this->b = b;
    }
        //成员函数
        CTest AddVale(CTest &T)
    {
        CTest Tmp(this->a + T.a, this->b + T.b);
        return Tmp;
    }
    //返回函数的引用，是返回本身
```

```
    CTest &TestAdd(CTest &T)
    {
        this->a = this->a + T.a;
        this->b = this->b + T.b;
        return *this;
    }
    void Print()
    {
        cout << "a = "<<a<<"  "<<"b = "<<b<<endl;
    }

};
    //全局的函数

CTest AddVale(CTest &t1,CTest &t2)
{
    CTest RCTest(t1.a + t2.a,t1.b + t2.b);
    return RCTest;
}
    //将成员函数转化为全局函数
void Print(CTest *PT)
{
    cout << PT->a << "   "<< PT->b << endl;
}
void main_07()
{
    CTest t1(1,3);
    CTest t2(5,3);
    //两个对象之间的运算
    CTest T = t1.AddVale(t2);  //对 T 进行初始化
    T.Print();
    CTest T1(5,6);
    T1 = t1.AddVale(t2);  //对 T1 复制
    T1.Print();
    CTest T2(1,6);
    T2 = AddVale(t1,t2);
    T2.Print();
```

```
    //返回引用
    // t2 = t2 + t1;
    T2.TestAdd(T1);
    T2.Print();

    system("pause");
}
/*************************************************************************/
/*
    友元函数可以访问类的私有成员，是全局函数，friend void FriendFun(class *p,datatype);
    友元函数声明的位置和 private、public 无关
    友元类的使用，B 类是 A 类的友元类，则 B 类可以访问 A 类的所有成员,说明 A 是 B 的子属性
    友元类破坏了类的封装性
*/
/*************************************************************************/
class A
{
private:
    int a;
    int b;
    friend class B; //B 可以访问 A 是私有成员
public:
    A(int a = 0,int b = 0)
    {
        this->a = a;
        this->b = b;
    }
    //友元函数声明
    friend void Modify(A *p,int var);
    int GetA(){return this->a;}
}
void  Modify(A *p,int var)
{
    p->a = var;

}
```

```
class B
{
private:
	A object;
public:
	void SetA(int var)
	{
		object.a = var;
		cout << object.a <<endl;
	}
};
void main_08()
{
	A a(2,3);
	B b;
	b.SetA(100);
	system("pause");
}
/*************************************************************************/
/*
```

运算符重载：

有两种方法：全局函数和成员函数

运算符的重载基础：

(一元运算符重载)（二元运算符重载）前置，后置

运算符重载使得用户自定义的数据以一种更简洁的方工作

函数重载就是对一个已有的函数赋予新的含义，使之实现新的功能，因此一个函数名就可以用来代表不用功效的函数

运算符重载机制：

运算符函数是一种特殊的成员函数或者友元函数

成员函数的语法是：

类型类名:operator op (参数列表){ }

（返回值） （关键字） （函数名） （操作数）

一个运算符被重载后，原有的意义没有失去，只是定义了相对以特定类的一个新运算符

例如：Complex operator+(Complex & c1, Complex & c2); //全局函数

Complex operator-(Complex &c2); //类成员函数

运算符的重载可以对运算符作出新的解释，单数原有的语义不变

不改变运算符的优先级、结合性

不改变运算符所需要的操作数

不能创建新的运算符

用成员或者友元函数重载运算符：

（1）运算符重载可以重载为成员函数或者友元函数

（2）关键区别在成员函数具有 this 指针，友元函数没有 this 指针

（3）不管是成员函数还是友元函数重载，运算符的使用方法相同

（4）但是传递参数的方式，实现的代码、应用场合不同

定义运算符重载函数名的步骤

全局函数，类成员函数实现运算符重载步骤：

1）承认操作符重载是个函数，写出函数名称

2）根据操作数，写出函数参数

3）根据实际情况，完善函数返回值，以及实现目的

运算符实现的两种方法：友元函数（全局函数）+ 类成员函数

在运算符重载的过程中，需要判断函数返回值的类型（函数本身（使用引用）、返回元素、返回指针）

等号操作符的重载步骤：（将对象旧的内存释放掉、开辟新的内存，返回的是函数的引用）

函数返回值当左值，需要使用引用

a++ : 先使用旧值运算，再+1

++a :先自增再+1

++a ;返回的是本身

a++ : 返回的是值

/**********运算符重载基础**********/

/*c3 = c1 + c2;

Complex 也是一种数据类型，是用户自定义的，要想实现对象间的运算，需要重载运算符

1 全局运算符重载

Complex operate+(Complex &c1,Complex &c2);

2 成员函数重载 v1.operate+(v2);

```
        Complex operate+(Complex &c1)
        */
        //二元运算符重载
class Complex
{
private:
        int a;
        int b;
public:
        Complex(int a = 0,int b = 0)
        {
                this->a = a;
                this->b = b;
        }
        Complex (Complex &Obj)
        {
                this->a = Obj.a;
                this->b = Obj.b;
        }
        ~Complex() { }
        void PrintComplex()
        {
                cout << a <<"+"<<b<<"i" <<endl;
        }
public:
        //二元运算符的重载
        //全局函数
        friend Complex operator+(Complex &c1,Complex &c2);
        friend Complex MyAdd(Complex &c1,Complex &c2);
        //成员函数
        Complex operator-(Complex&c)
        {
                Complex RC(this->a - c.a,this->b - c.b);
                return RC;
        }
public:
        //一元运算符的前置重载
        //全局函数
```

```
    friend Complex &operator++(Complex &c);
    //成员函数
    Complex operator--()
    {
        this->a--;
        this->b--;
        return *this;
    }
public:
    //一元运算符的后置处理
    //后置++
    friend Complex &operator++(Complex &c,int);
    //成员函数后置--
    Complex operator--(int)
    {
        Complex tmp = *this;
        this->a--;
        this->b--;
        return tmp;
    }
    //左移操作符的重载
    friend ostream &operator<<(ostream &out,Complex&var);
    ostream &operator<<(ostream &out)
    {
        out << this->a<<" + "<<this->b<<"i"<<endl;
        return out;
    }
};
    //1 定义了全局函数
Complex MyAdd(Complex &c1,Complex &c2)
{
    Complex tmp(c1.a + c2.a,c1.b + c2.b);
    return tmp;
}
    //全局运算符重载函数
Complex operator+(Complex &c1,Complex &c2)
{
    Complex tmp(c1.a + c2.a,c1.b + c2.b);
```

```
	return tmp;
}
	//全局函数
Complex &operator++(Complex &c)
{
	c.a++;
	c.b++;
	return c;
}
Complex &operator++(Complex &c_var,int)
{
	Complex tmp = c_var;
	c_var.a++;
	c_var.b++;
	return tmp;
}
	//全局函数
ostream &operator<<(ostream &out,Complex&var)
{
	out << var.a<< " + "<< var.b<<"i"<<endl;
	return out;
}
void main_09()
{
	int a = 1,b = 2;
	int c;
	c = a + b;
	//基础数据类型编译器已经知道，如何运算
	Complex c1(2,3);
	Complex c2(5,6);
	Complex c3 = MyAdd(c1,c2);
	c3.PrintComplex();
	//步骤 2
	Complex c4 = c1 + c2;
	c4.PrintComplex();

	Complex C5 = c4 - c1;
	C5.PrintComplex();
```

```
	//前置++操作符（）全局函数
	Complex C6(8,9);
	++C6;
	C6.PrintComplex();
	//前置--操作符，成员函数
	Complex C7(9,9);
	--C7;
	C7.PrintComplex();

	//后置++
	Complex C8(6,9);
	C8++;
	C8.PrintComplex();

	cout << C8;
	cout << C7;
	system("pause");

	//如何重载左移操作符
	//总结：（1）运算符的重载本质上是函数的调用
}
	//括号运算符的重载
class F
{
public:
	int operator()(int i,int j)
	{
		return i*i+j*j;
	}
};
void main_1()
{
F f;
f(2,6);//这种写法对应两种格式：1）对象的申明调用有参构造函数，2）函数的重载
	//重载方式
	}
/*************************************************************************/
/*
```

逻辑运算符&&（与）和逻辑运算符||（否）

理论知识：

为什么不要重载&&和||

1)&&（与）和||（否）是 C++中非常特殊的运算符

2)&&和||内置实现了短路

3)操作符的重载是靠函数重载实现完成的

4)操作数作为函数参数传递

5)C++的函数参数都会被求值，无法实现短路

&&和||可以实现重载，但是无法实现短路规则，因此一般情况下不对着两个运算符进行重载

*/

/***/

//实现一个字符串类

//构造函数函数的要求：MyString a; MyStrring("absc");MyString b = a;

//常用的操作符：<< >> != == < > =

//程序在 Cpogram 工程中查阅

/***/

/*

类的继承：

面向对象的程序设计有四个主要的特点：抽象，封装，继承和多态性；

has-A,uses-A 和 is-A 的区别：

has-A：包含关系，用以描述一个类是由多个部件类构成，实现 has-A 关系用类成员表示，即一个类中的数据成员是另一个一定定义的类

uses-A:一个类部分使用另一个类，通过类之间成员函数的相互关系，定义友元或者对象参数传递实现

is-A:机制称为继承，关系具有传递行，不具有对称性

继承的相关概念：

一个 B 类继承 A 类，或者从 A 类派生类 B：A：基类（父类），B:派生类（子类）

类继承关系的语法形式：

class 派生类名：基类名表

{

数据成员和成员函数的申明

}

基类名表的构成：访问控制：基类名 1，访问控制：基类名 2...访问控制 基类名 n

访问控制表示派生类对基类的继承方式： 共有继承，私有继承，保护继承

注意：C++中继承的访问方式会影响子类对外访问的属性

继承的重要说明：

子类拥有父类所有成员变量和成员函数

子类是一种特殊的父类

子类可以当做父类对象使用

子类可以拥有父类没有的方法和属性

派生类的访问控制：

派生类继承了基类的全部成员表变量和成员方法（除了构造和析构之外的成员方法），但是这些成员的访问属性，在派生过程中可以调整

不同的继承方式会改变继承成员的访问属性

public 继承：父类成员在子类中保持原有的访问级别

private 继承：父类在子类中变为 private 成员

protected 继承：父类中的 public 成员变为 protected，在派生类内部使用

父类 protected 成员变为 protected

父类的 private 成员仍然为 private

private 成员早子类中依然存在，但是却无法访问控制，不论何种方式继承基类，派生类都不能直接使用派生类的基类

public 继承：在类的内部和外部都能使用

protected 继承：在类的内部使用，在派生类的内部使用

继承中的构造和析构

类型的兼容性原则：

是指在需要基类对象的任何地方，都可以使用公有派生类的对象替代。通过公有继承、派生类得到了基类中除了构造函数、析构函数以外的所有函数，这样，公有派生类实际上具有了基类的所有功能，凡是基类能解决的问题，公有派生类都可以解决。

类型的兼容性包括以下情况：

1）子类对象可以当做父类对象使用

2）子类对象可以直接赋值给父类对象

3）子类对象可以直接初始化父类对象

4）父类指针可以直接指向子类对象

5）父类引用可以直接引用子类对象

在替代之后，派生类对象可以作为基类的对象使用，但是只能使用从基类继承的成员。类型的兼容性规则是多态性的重要基础。

如何初始化基类成员？基类构造函数和派生类的够着函数有什么关系？

在派生类对象构造时候，需要调用基类的构造函数对其继承来的成员进行初始化

在派生类的析构中，需要调用基类的析构函数对其继承的成员进行清理。

//有参构造函数

Parent(int a,int b)//基类的构造函数

Child(int a,int b, int c):Parent(a,b)

初始化：Child c(1,2,3) ：先进入基类的构造函数-》派生类的构造函数-》派生类的析构—》基类析构函数

当基类的构造函数是有参构造函数时候，需要在派生类的初始化列表中显示调用

继承与组合混搭情况下，构造和析构调用原则：

先构造基类，再构造成员变量，最后构造自己

先析构自己，再析构成员变量，最后析构基类

继承中同名成员变量处理方法：

1.当子类的成员变量和父类的成员变量同名时候，子类依然从父类继承同名变量

2.在派生类中使用基类的同名成员，显式地使用类名限定符

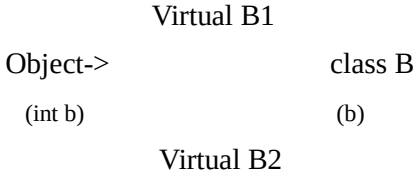
派生类中的 Static 关键字

基类定义的静态成员，将被所有派生类共享

根据静态成员自身访问特性和派生类的继承方式，在类层次体系中具有不同的访问性质

派生类中访问静态成员，使用类的显式进行调用

对于继承过程中产生的二异性：则引入虚继承 Virture



class B1: virtual public Object

*/

/***/

/*

多态：同样的调用语句，有多种不同的表现形式

当派生类和基类中出现相同名的成员函数时；

编译器的做法不是我们期望的

根据实际的对象类型来判断重写函数的调用

如果基类指针指向的是基类对象则调用基类成员常数

如果基类的指针指向的是派生类的对象则调用派生类中的重写函数

解决方案：

C++中的多态支持

C++通过 virtual 关键字对多态进行支持

使用 virtual 声明的函数被重写即可展现多态性

封装突破了函数的概念，用类做函数的参数。可以使用对象的属性和对象的方法

继承：可以复用代码

多态：不仅具有继承，还可以是适应未来

间接赋值是指针的最大意义：

1)两码事：指针变量和它指向的内存块变量

2)条件反射：指针指向某个变量，就是把某个变量的地址给了指针

3)间接赋值的 3 个条件：（1）2 个变量（实参，形参）（2）建立关系：实参取地址给形参指针（3）*p 形参去间接修改实参的值

间接赋值是 C 语言特有的现象

实现多态的三个条件：

（1）要有类的继承

（2）要有成员函数重载

（3）使用基类指针指向派生类对象

多态的理论基础：

静态联编和动态联编

1.联编是指一个程序模块、代码之间互相关的过程

2.静态联编是程序的匹配，连接在编译阶段实现，也成为早期匹配　程序在编译的时候就决定了怎么执行

3.动态联编是指程序联编推到到程序运行时进行，所有又称为晚期联编,switch 和 if 语句是动态联编的例子，程序在运行的时候才决定调用什么函数

4.理论联系实际

C++和 C 相同，是静态编译语言

在编译时，编译器会自动根据指针的类型判断指向的是一个什么样的对象，所以编译器认为父类指针指向的是父类对象

由于程序没有运行，所以不知道基类指针指向的对象是基类对象还是派生类对象，从程序安全角度来说，编译器假设基类指针指向的是基类对象，因此编译的结果为调用基类的成员函数，这种特性就是静态联编

多态的 C++实现：

virtual 关键字，告诉编译器这个函数要支持多态，不要根据指针类型判断如何调用，而是要根据指针所指向的实际对象类型来判断如何调用，冠以 virtual 关键字的函数叫做虚函数，虚函数分为两类：一般虚函数和纯虚函数，

```
    不写 virtual 关键字。则是静态联编
    多态：同样的调用语句有多种不同的变现形式
*/
/*************************************************************************/
class Parent
{
public:
    Parent(int a)
    {
        this->a = a;
    }
public:
    //在积累中加 virtual
    virtual void Print()
    {
        cout << "a="<<a<<endl;
    }
private:
    int a;
};
class Child :public Parent
{
public:
    Child(int a,int b): Parent(a)
    {
        this->b = b;
    }
public:
    void Print()
    {
        cout <<"b="<<b<<endl;
    }
private:
    int b;
};
void How2Print(Parent *p)
{
    p->Print();  //一种调用形式，多种表现形式
```

```
}
void main_2()
{
	Child C(1,2);
	Parent *p = NULL;
	Parent P(20);
	p = &P;
	How2Print(p);
	p = &C;
	How2Print(p); //调用的是基类的成员函数
	system("pause");
}

//多态案例：
class HeroFighter
{
public:
	virtual int power(){return 10;}
};
class advHeroFighter :public HeroFighter
{
public:
	virtual int power(){return 20;}
};
class EnemyFighter
{
public:
	int attack(){return 15;}
};
//对象的舞台  看成一个框架，
void Fight(HeroFighter *pHF,EnemyFighter *pEF)
{
	//基类指针可以指向基类对象，也可以指向派生类对象
	if(pHF->power() > pEF->attack())
		cout <<"主角 WIN"<<endl;
	else
		cout <<"敌机 WIN"<<endl;
}
```

```
void main_()
{
	HeroFighter HF;
	EnemyFighter EF;
	advHeroFighter aHF;
	Fight(&HF,&EF);
	Fight(&aHF,&EF);
	system("pause");
}
//没有使用多态的做法
void main__()
{
	HeroFighter HF;
	EnemyFighter EF;
	advHeroFighter aHF;
	if(HF.power()>EF.attack())
		cout <<"主角 WIN"<<endl;
	else
		cout <<"敌机 WIN"<<endl;
	if(aHF.power()>EF.attack())
		cout <<"adv 主角 WIN"<<endl;
	else
		cout <<"敌机 WIN"<<endl;
	system("pause");
}
/*************************************************************************/
/*
	虚析构函数：
	在什么情况下声明虚析构函数：
	1）构造函数不能使虚函数，建立一个派生的类对象的时候，必须从类层次的根开始，沿着继承路径逐个调用基类的构造函数
	2）析构函数可是是虚的，析构函数用于指引 delete 运算符正确析构动态对象
	普通的析构函数在删除动态派生类对象的调用情况：
	析构由基类指针建立的派生类对象，没有调用派生类的析构函数,无法对派生类对象进行析构
*/
/*************************************************************************/
```

```
class AMODEL
{
public:
    AMODEL()
    {
        p = new char[20];
        strcpy(p,"Amodel");
        cout << "调用 A 类构造函数"<<endl;
    }
    virtual ~AMODEL()
    {
        delete []p;
        cout << "调用 A 类析构函数"<<endl;
    }
private:
    char *p;
};
class BMODEL : public AMODEL
{
public:
    BMODEL()
    {
        p = new char[20];
        strcpy(p,"Amodel");
        cout << "调用 b 类构造函数"<<endl;
    }
    ~BMODEL()
    {
        delete []p;
        cout << "调用 b 类析构函数"<<endl;
    }
private:
    char *p;
};
class CMODEL : public AMODEL
{
public:
    CMODEL()
```

```
    {
        p = new char[20];
        strcpy(p,"Amodel");
        cout << "调用 C 类构造函数"<<endl;
    }
    ~CMODEL()
    {
        delete []p;
        cout << "调用 c 类析构函数"<<endl;
    }
private:
    char *p;
};
void howTodelete(AMODEL *m_P)
{
    delete  m_P;
}
void main_20()
{
    CMODEL * MyC = new CMODEL;  //派生类对象初始化的时候，先调用基类的构造函数，再调用派生类的构造函数
    howTodelete(MyC);  //普通情况下，在析构的过程中，将派生类对象传给基类指针，在析构的过程中，只调用了基类的构造函数，只能对基类的对象进行析构，不能对派生类对象进行析构，没有体现书多态
    //如果想通过基类指针，释放所有的资源(调用所有类的析构函数)，则需要使用虚析构函数，在基类构造虚析构函数
    system("pause");
}

/*****************************************************************************/
/*
    重写 PK 重载理解
    1) 函数重载：
    必须在一个类中进行；派生类无法重载基类的函数，基类的同名函数将被覆盖；重载是在编译期间根据参数的类型和个数决定函数调用
    2)函数重写：
    必须发生在基类和派生类之间的；并且基类和派生类中的函数必须有完全相同的
```

函数原型；使用 virtual 声明之后能够产生多态，如果不使用 virtual，则叫重定义；多态是在运行期间根据具体对象的类型决定函数的调用

*/

/**/

/**/

/*

多态的原理研究：

理论知识：

1)当类中声明虚函数，编译器会在类中会生成一个虚函数表

2)虚函数表是一个存储类成员函数指针的数据结构

3)虚函数表是由编译器自动生成和维护的

4)virtual 成员函数会被编译器放入到虚函数表中

5)当存在虚函数时，每个对象中都有一个指向虚函数表的指针(C++编译器给基类对象、派生类对象提前布局 vptr 指针，当进行 howtoPrint(Parent *base))函数的时候，C++编译器不需要区分基类对象和派生类对象，只需要在 base 中找到 vptr 即可

6)vptr 一般作为类对象的第一个成员

说明：

1)通过虚函数表指针 vptr 调用重写函数是在程序运行的时候进行的，因此需要寻地址操作才能确定真正应该调用的函数，而普通的函数在编译的时候就完成了函数的调用，在效率上，虚函数的效率要低很多

2)出于效率的考虑，没有必要将所有的成员设置为虚函数。

3)C++编译器不需要区分基类对象和派生类对象，是通过 vptr 调用对应对象的对应函数

面试题 1:谈谈你对多态的理解：

多态的实现效果：同样的调用语句具有不同的表现形式。

多态实现的调节：继承，virtual 重写函数，基类指针指向派生类对象

多态的 C++实现： virtual 关键字，告诉编译器这个函数要支持多态，不要根据指针类型判断如何调用，而是要根据指针所指向的实际对象类型来判断如何调用，冠以 virtual 关键字的函数叫做虚函数，虚函数分为两类：一般虚函数和纯虚函数。

多态的理论基础：动态联编 PK 静态联编，根据实际的对象类型来判断调用的函数。

多态的重要意义：设计模式的基础

基类指针和派生类指针步长不一样。

多态是用父类指针指向子类对象和父类步长++是两个不同的概念。

什么时候子类步长和父类步长一样？

指针的铁律 1：

1）指针是一种变量，占用内存空间，用来保存内存地址,测试指针变量占有内存空间大小。

2）*p 操作内存，指针申明的时候，*号表示所声明的变量为指针，在指针使用的时候，*表示指针所指向的内存空间的值。

*p 相当于通过地址（p 变量的值）找到一块内存，然后操作内存，*p 放在等号的左边赋值，*p 放在右边为取值。

3）指针变量和它所指向的内存块是两个概念。

含义 1：给 p 赋值 p= 0x1111,只会改变指针变量值，不会改变所知的内容。

含义 2：*p='a',不会改变指针变量的值，只会改变所指向的内存块的值。

4）指针是一种数据类型，是指它指向的内存空间的数据类型。

含义 1：指针步长（p++），根据所指向的内存空间的数据类型来确定。

注意：建立指针指向谁，就把谁的地址赋给指针，不断的给指针变量赋值，就是不断的改变指针的指向，和所指向的内存空间没有任何关系。

*/

/***/

/***/

/*

纯虚函数和抽象类：

纯虚函数是一个在基类中说明的虚函数，在基类中没有定义，要求在任何派生类中都定义自己的版本。

纯虚函数为各派生类提供了一个公共界面（接口的封装和设计，软件的模块功能划分）

纯虚函数的基本形式：

virtual 类型 函数名（参数表）= 0；

一个具有纯虚函数的类是一个抽象类。

抽象类不能建立对象，可以声明抽象类的指针；抽象类不能作为返回类型，不能作为参数类型，但是可以声明抽象类的引用。

*/

/***/

```
//面向抽象类编程（面向一套先定义好的接口编程）
//解耦合.....模块的划分
class Figure
{
public:
    virtual int getArea() = 0;//纯虚函数
```

```
};
class Circle : public Figure
{
public:
	Circle(int r){m_r = r;}
	virtual int getArea(){return 3.14*m_r*m_r;}
private:
	int m_r;
};
class FRect : public Figure
{
public:
	FRect(int a,int b)
	{
		this->a = a;
		this->b = b;
	}
	virtual int getArea(){return a*b;}
private:
	int a;
	int b;
};
class Triangle : public Figure
{
public:
	Triangle(int a,int h)
	{
		this->a = a;
		this->h = h;
	}
	virtual int getArea(){return 0.5*a*h;}
private:
	int a;
	int h;
};
void HowToPrint(Figure *f)
{
	int Area = f->getArea();
```

```
    cout <<"Area="<<Area <<endl;
}
void main_21()
{
    //Figure f;//抽象类不能建立对象
    Figure *f = NULL; //可以声明抽象类的指针
    Circle c(5);
    FRect fr(5,8);
    Triangle t(5,8);
    f = &c;
    int fR = f->getArea();
    cout << fR <<endl;
    HowToPrint(&c);
    f = &fr;
    int fR1 = f->getArea();
    cout << fR1 <<endl;
    HowToPrint(&fr);
    f = &t;
    int ft = f->getArea();
    cout << ft <<endl;
    HowToPrint(&t);
}
/*************************************************************************/
/*
```

抽象类在多继承中的应用：

C++没有 java 中的接口概念，抽象类可以模拟 java 中的接口类（接口和协议）

有关继承的说明：

1.工程上的多继承：

1)工程开发中真正意义上的多继承是几乎不被使用的，多继承带来的代码复杂性远多于其带来的便利。

2)多继承对代码维护性上的影响是灾难性的。

3)在设计方法上，任何多继承都可以用单继承实现

2.多继承中的二义性和多继承不能解决的问题

3.多继承的应用场景：

1)绝大多数面向对象语言都不支持多继承

2)绝大多数面向对象的语言都支持接口的概念

3)C++中没有接口概念

接口类中只有函数原型定义，没有任何数据的定义

```
class InterFace
{
   public:
       virtual void Fun0()=0;
       virtual void Fun1(int i)=0;
       virtual void Fun2(int j,int k)=0;
};
```

多重继承接口不会带来二义性和复杂性等问题，多重继承可以通过精心设计来单继承和接口代替。

接口类只是一个功能说明，而不是功能实现。

子类需要根据功能说明定义功能实现

C++可是使用纯虚函数实现接口

*/

/***/

```
//抽象类在多继承中的应用
class InterFace
{
public:
    virtual int iSum(int a,int b) = 0;
    virtual void Print() = 0;
};
class mInterFace
{
public:
    virtual int iMulti(int a,int b) = 0;
    virtual void  Print() = 0;
};
class IParent
{
public:
    int getA(){return a;}
private:
    int a;
};
class IChild : public InterFace,public mInterFace,public IParent
```

```
{
public:
	virtual int iSum(int a,int b) {return a+b;}
	virtual void Print()
	{
		cout << "Sum=0"<<endl;
	}
	virtual int iMulti(int a,int b) {return a*b;}
};
void main_22()
{
	IChild C;
	InterFace *m_p = &C;
	mInterFace *n_p = &C;
	m_p->iSum(2,3);
	n_p->iMulti(2,3);
}
/*****************************************************************************/
/*
	//面向对象的重要思想：
	组合和继承：框架构建，采用组合的方式较好
	投入
	控制反转
	aop 编程是对继承的有利补充
*/
/*****************************************************************************/
/*****************************************************************************/
/*
	数组类型的基本语法
	定义一个数组类型，定义一个指针数组类型，定义一个指向数组类型的指针
	定义数组类型：typedet int(MyArrayType)[10];  MyArrayType MyArray;MyArray[0];
	定义一个指针数组类型: typedet(*MyArrayType)[10];MyArrayType MyArray;
	函数指针的基本语法;定义一个函数类型，定义一个函数指针类型，定义一个指向函数指针
	int add(int a ,int b)
	{
	  return a+b;
```

```
}
typedef int (MyFuncType)(int a,int b);//定义一个函数类型

MyFuncType *p_Func = NULL;  //定义了一个指针，指向某一种类的函数

p_Func = &add; //取地址

p_Func(2,3);//间接调用函数

定义一个函数指针类型： typedef int (*MyFuncType)(int a,int b);//定义一个函数类型

MyFuncType p_Func；//定义了一个指针
p_Func = add;
p_Func(52,78);

//函数指针
int (*MyFuncType)(int a,int b);//定义一个变量
MyFuncType = add;
MyFuncType(52,36);
```

函数指针做函数参数的思想：

函数的指针类型：typedef 函数返回值（*MyFuncType）（参数列表）

C++编译器支持多态是通过提前布局（cvtr 和虚函数表）

请问： C 编译器到底是通过哪个具体的语法，来实现接耦合的

结论：回调函数的本质：提前做了一个协议的约定（把函数的参数，函数的返回值提前约定）

C 面向接口编程 C 多态

函数指针的用法、函数指针做函数参数的思想剖析、函数指针在开发中的实战

dll 利用编译器调用使用；Win32 环境下动态链接库 dll 编程原理

```
*/
/*************************************************************************/
//利用函数指针实现函数的多态，是回调函数的案例
int add(int a,int b)//任务的实现
{
    return a+b;

}
```

```
int iSum(int a,int b)
{
	return a+b;

}
//函数指针作为函数参数
//定义一个类型
typedef int(*MyFuncType)(int a,int b);

int MainOP(MyFuncType p_Func)//任务的调用
{
	int Result = p_Func(6,8);
	cout <<"MainOP Function Result = "<<Result<<endl;
	return Result;
}
int MainOP2(int(*MyFuncType)(int a,int b))//任务的调用
{
	int Result = MyFuncType(5,6);
	cout <<"MainOP2 Function Result = "<<Result<<endl;
	return Result;
}
void main_3()
{
	add(5,6);//直接调用
	MyFuncType p_Func = NULL;
	p_Func = add;
	int Result = p_Func(8,9);
	cout << "Result="<< Result<<endl;
	MainOP(p_Func);
	MainOP2(p_Func);
	MainOP(iSum);
	MainOP(add);
	MainOP2(iSum);
	MainOP2(add);

	//这样做可以实现任务的调用和任务的实分开；
	system("pause");
}
```