C++面向对象，内容围绕类进行讲解

类，类成员，默认函数，自定义成员函数，访问权限，

继承，多继承，虚继承，多态

运算符重载，友缘，动态申请，类型转换

string，文件流

C++是面向对象的语言？

这是不对的，C++有面向过程的语法基础，有面向对象的语法基础，有泛型编程的语法基础，有函数式编程的语法基础，这是四个大类编程思想，所以 C++是非常强大的。而且有专业的 STL，BOOST 库，非常狠。

有很多人说 C++难学，主要原因就是这四部分内容混到一起学了。本来知识点就极多，各种思想混到一起学，根本分不清。我当时就是，学完了脑子里乱七八糟，一大堆语法，完全没有章法。

面向过程，面向对象

面向过程与面向对象不是语法，是编程思想，编程思路。

编程就是处理数据，所以两者的区别是对于数据结构与其算法函数之间的设计思想与使用思想的区别。即编程思想上的区别。

编程思想，也就是写代码的思路。思路是不取决于语言的，C 语言也能写面向过程思想的代码，也能写面向对象思想的代码，也能实现泛型编程思想的代码，也能封装函数式编程思想的代码。

举例说明面向过程与面向对象的具体区别

1、数据变量与处理函数之间的互相独立

面向过程开发，数据与算法函数是分割的，没有直接关系，比如链表，我们用一个 head1 变量表示链表，初始化链表函数 init 可以对 head1 链表初始化，也可以对另外的 head2 链表初始化，还能给 head3 链表初始化。互相不是专属的，此外还有释放函数 free，也是如此，而且 init 与 free 也是没有任何关系。

面向对象开发，将数据变量与处理函数绑定在一起，成为一体，即数据自带专属处理函数，构成一个系统，此时的数据就是一个对象，对象包含数据本身也包含数据处理函数，逻辑上：定义链表对象 head1，自带 init 与 free，定义链表对象 head2，自带 init 与 free，定义链表对象 head3，自带 init 与 free。

## 2、数据变量与处理函数自动调用与手动调用

面向过程开发，需要什么功能，我就得调用什么函数，head1 需要初始化我得自己调用 init，传递参数 head1(因为二者互相独立的)，在需要释放的时候调用 free，传递参数 head1(因为二者互相独立的)。head2，head3 也是如此。互相独立。属于，需要什么功能，就调用什么函数，流水线的过程，一步也不能少。

面向对象开发，函数与数据是一体的，对象 head1 一创建，自动调用初始化，我们不用主动调用 init，对象生命周期结束，自动调用 free，而且不需要传递参数 head1。因为大家是一个系统，自由的互相使用。

### 那为什么说 C++面向对象，而 C 语言是面向过程呢？

是因为 C++专门设计了面向对象开发的专属语法，而 C 语言里没有，所以 C++写面向对象的代码就非常的容易，非常的快乐。C 语言里没有专门的，所以要实现面向对象就很麻烦，写的东西逻辑也不容易理解。

专门的语法 class，就是上面说的那堆，C 语言没有专门的。

### 如何更好的理解使用面向对象的编程思想？

需要大量的实践。实践多了，自然就能体会到其中的妙处，也即孰能生巧，想要通过一些大佬的嘴，指点几句，短时间就能成为编程高手，那是痴人说梦，痴心妄想，不可能的。

回归到编码，也就是把 class 相关的这套知识写到滚瓜烂熟，不停的写，写的越多经验越多，经验越多技巧就越多，技巧越多用的越帅，用的比别人都帅，那你的面向对象思想就更加的完美。

类：

什么是类？

从编码角度，类的语法是 class，就是一种数据类型，跟 struct，int，double 一样，然后用他定义变量，用 class 这种类型定义的变量叫对象，为什么呢？因为 class 这种类型的组成极其丰富，使得变量不仅仅是装数据的功能，还自带处理数据的功能，比 int a 这种变量强大一万倍，所以给他叫对象，比如小华，就是人类的一个个体，也叫对象。

class 也是 struct 功能的拓展延伸，struct 用来装纯数据，class 里可以装函数访问权限以及各种新功能。

抽象概念层面

具有相同属性与功能的对象的合集，叫类。比如人类，昆虫类，汽车类，飞机类，都是各自的一个群体，叫类，单独拿出来一个具体的个体，叫对象

概念性的文字，我们不用过分深究，把 class 相关的语法，用的非常熟练，就越来越能体会面向对象的设计思想了，不是一个月两个月能融会贯通的。

一段简单的类与对象代码

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

class Test   //class 关键字，Test 类名，一般首字母大写
{
private:    //私有访问权限，下一节单独讲
    int a;
public:    //公共访问权限
    void Print()  //输出数据
    {
        cout << a << endl;
    }
    int Get()  //获取数据
    {
        return a;
    }
};

int main(void)
{
    Test t;
    t.Print();        //自带输出
    int n = t.Get();  //获取对象数据
    cout << n << endl;

    return 0;
}
```

三个：

```
private    //私有的，只能在所在类内调用
protected  //受保护的，只能在所在类以及子类中使用，继承部分讲解
public     //公有的，可以在类内外需要的位置调用
```

书写格式：

```
private:
protected:
public:    //冒号
```

作用范围：从关键字以下，直到下一个关键字或者类结尾

最上方不写，默认是 private

```
class Test
{
    int c;  //默认是 private
    int a;  //默认是 private
public:    //公共访问权限
    void Print() { cout << a << endl; }
    int  Get()   { return a; }
};
```

数量不限制，顺序不限制，可以用来分类使用

```
class Test
{
private:       //整型变量
    int c;
    long a;
private:       //指针变量
    double* p1;
    float* p2;
public:        //公共访问权限
    void Print()     { cout << a << endl; }
    int  Get()       { return a; }
    void Set(int v) { a = v; }
private:       //非公共接口函数，类内部逻辑调用
    void Init() {}
    void Free() {}
```

```
    void Sort() {}

public:  //空格也行
    void Add() {}
    void Del() {}
    void End() {}
};
```

类内声明，类外定义

```
#include <iostream>
using namespace std;
class Test    //放头文件就行
{
private:
    int a = 23;
public:
    void Print();   //类内声明
    void Fun(int c = 7, int d = 3); //默认参数在声明指定
};
int main(void)
{
    Test t;
    t.Print();
    t.Fun();
    return 0;
}
void Test::Print()  //类外定义  Test:: 类名作用域
{
    cout << a << endl;
}
void Test::Fun(int c, int d) //类外定义
{
    cout << c << ' ' << d << endl;
}
```

构造函数

对象创建时自动调用的一个函数，用来对对象成员进行赋值等操作

形式：

```
class Test
{
private:
    int a = 23;
public:
    Test()    //构造函数，无返回值，名字是类名
    {
        cout << "构造函数" << endl;
    }
};
int main(void)
{
    Test t;        //自动调用构造函数
    Test* p;       //这里不调用，还没有对象构造函数
    p = new Test; //这里调用构造函数
    return 0;
    }
```

类内声明，类外定义，如下：

```
class Test
{
private:
    int a = 23;
public:
    Test();
};
Test::Test()    //无返回值，名字是类名
{
    cout << "构造函数" << endl;
}
```

由于其是对象创建时自动调用的，所以一般用来对对象成员做数据初始操作，比如数据成员赋值，比如调用初始功能的成员函数

注意：构造函数需要 public，否则不能直接创建对象，因为对象创建时，系统自动调用构造函数，如果定义成 private/protected，系统就没法调用了

构造函数赋值：

```
class Test
{
private:
    int a = 23;
public:
    Test()   //无返回值，名字是类名
    {
        a = 34;  //变量 a 赋值，不叫初始化
        cout << "构造函数" << endl;
    }
    void Print()
    {
        cout << a << endl;
    }
};

int main(void)
{
    Test t;     //自动调用构造函数
    t.Print();  //输出 34，原来的 23 被覆盖了

    return 0;
}
```

默认构造函数

当我们不写构造函数时，系统会默认有一个构造函数，叫默认构造函数

其为：

```
Test()    //无参数，无代码，手写构造函数后，就没有默认构造了
{
}
```

有参数的构造函数

```
class Test
{
private:
    int a = 23;
public:
    Test(int aa)    //无返回值，名字是类名
    {
        a = aa;      //用参数对其赋值
        cout << "构造函数" << endl;
    }
    void Print() { cout << a << endl; }
};

int main(void)
{
    Test t(12); //构造有参数，必须得传参数，不然报错
    t.Print();  //输出 12
    return 0;
}
```

初始化列表　　引用成员，const 成员必须使用此

```
class Test
{
private:
    int a = 23;
    float f;
public:
    Test(int aa, float ff) : a(aa) , f(ff)
              //冒号，多个用逗号隔开，aa 给 a 初始化,ff 给 f 初始化
    {
    }
    void Print() { cout << a << ' ' << f << endl; }
};

int main(void)
{
    Test t(12, 2.3f); //构造有参数，必须得传参数，不然报错
    t.Print();  //输出 12，2.3
    return 0;
}
```

类外定义位置：

```
class Test
{
private:
    int a = 23;
    float f;
public:
    Test(int aa, float ff); //声明不放初始化列表
    void Print() { cout << a << ' ' << f << endl; }
};
Test::Test(int aa, float ff) : a(aa), f(ff)  //放定义处
{
}
```

与构造赋值的区别

构造函数内叫赋值，初始化列表叫初始化

举例：

```
Test(int aa, float ff) : a(aa) , f(ff)  //aa 给 a 初始化,ff 给 f 初始化
{
    a = 22;
    f = 33.3f;
}
int main(void)
{
    Test t(12, 2.3f); //构造有参数，必须得传参数，不然报错
    t.Print();  //输出 22,33.3

    int a = 12;         //初始化
    a = 33;             //赋值，是一个先后的问题
    cout << a << endl; //输出 33

    return 0;
}
```

实际应用中对于引用变量，const 变量这类必须初始化的成员，必须要用初始化列表，构造函数会报错

```
class Test
{
private:
    int& a;
    const float f;
public:
    Test(int& aa, float ff) : a(aa), f(ff)  //必须用这个
    {   //这个 aa 必须加引用      //a 间接引用外部的 c
        //如果 aa 不加引用，int aa; 那么 a 就是引用 aa
            //aa 是局部变量，会被释放，那不行
    }
    void Print() { cout << a << ' ' << f << endl; }
};
```

```
int main(void)
{
    int c = 12;
    Test t(c, 2.3f); //构造有参数，必须得传参数，不然报错
    t.Print();  //输出 12 2.3
    return 0;
}
```

与直接初始化的区别，两种初始化区别

```
class Test
{
private:
    int a = 12;
    int c = 22;
public:
    Test(int aa) : a(aa)  //初始化
    {
    }
    void Print() { cout << a << ' ' << f << endl; }
};

int main(void)
{
    int c = 12;
    Test t1(12); //a = 12  b = 22
    Test t2(22); //a = 22  b = 22
    Test t3(32); //a = 32  b = 22
    Test t4(42); //a = 42  b = 22
    Test t5(52); //a = 52  b = 22
    //每个对象 c 都是 22，没办法改初始值，a 可变的
    return 0;
}
```

构造函数重载

构造函数可以存在多个，创建对象时，只调用其中之一

多个构造函数之间是重载关系，即参数列表不一样

对象定义时，根据所传参数调用相应的构造函数

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

class Test
{
private:
    int   a = 0;
    float f = 0.0f;
public:
    Test(int aa, float ff) { a = aa, f = ff; }  // t3
    Test(float ff, int aa) { a = aa; f = ff; }  // t2
    Test()                 { }                  // t1
    Test(int aa)           { a = aa; }          // t4
    Test(float ff)         { f = ff; }          // t5
    void Print()           { cout << a << ' ' << f << endl; }
};
int main()
{
    Test t1;  // 无参数就不写小括号
    t1.Print(); // 0 0.0
    Test t2(2.5f, 2);
    t2.Print(); // 2 2.5
    Test t3(3, 4.5f);
    t3.Print(); // 4 4.5
    Test t4(2);
    t4.Print(); // 2 0.0
    Test t5(5.6f);
    t5.Print(); // 0 5.6f
    return 0;
}
```

默认参数的构造函数

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

class Test
{
private:
    int   a = 0;
    float f = 0.0f;
public:
    Test(int aa = 0, float ff = 0.0)
    {
        a = aa;
        f = ff;
    }
    /*
    Test(){  }不能与无参共存，不然 t1 就不知道调用哪个构造
    Test(int aa){ }不能与这个共存，不然 t2 就不知道调用哪个构造
    */
    void Print() { cout << a << ' ' << f << endl; }
};
int main()
{
    Test t1;
    t1.Print(); // 0 0.0
    Test t2(2);
    t2.Print(); // 2 0.0
    Test t3(3, 4.5f);
    t3.Print(); // 4 4.5
    return 0;
}
//好处，一个构造可以满足多种传参情况，更贴心
```

析构函数

对象生命周期结束时自动调用的函数形式：

```
#include <iostream>
using namespace std;
class Test
{
private:

    int a;
public:
    Test(int aa)
    {
        a = aa;
        cout << a << "构造函数" << endl;
    }
    ~Test() //无返回值，无参数，函数名为类名，前带符号~
    {
        cout << a << "析构函数" << endl;
    }
};
int main()
{
    Test t1(1);  //t1 开始

    {
        Test t2(2); //t2 开始
    } //t2 结束

    Test* t3;
    t3 = new Test(3); //t3 开始
    delete t3;  //t3 结束
    return 0;
} //t1 结束
```

对象数组的申请与释放

delete[] 对象数组，带方括号才能为每个对象调用析构函数

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

class Test
{
private:
    int a;
public:
    Test(int aa)
    {
        a = aa;
        cout << a << "构造函数" << endl;
    }
    ~Test() //无返回值，无参数，函数名为类名，前带符号~
    {
        cout << a << "析构函数" << endl;
    }
};
int main()
{
    Test* t = new Test[10]{1,2,3,4,5,6,7,8,9,10}; //必须得初始化
    //delete[] t; //只会调用：1 析构函数  后 9 个不会调用，并且会异常
    delete[] t; //可以调用所有对象的析构函数
    return 0;
}
```

this 指针

装着当前对象的地址

即每个对象都有自己的地址，我们可以通过 &t1 &t2 ... 获取到各个对象的地址，C++在类内直接提供了地址，我们就不需要在类外获取了

```
class Test
{
private:
    int a;
    Test* p = this;  //类内可以直接使用，这里没什么意义
    //Test* this; 相当于类内有个这，C++自动实现了对其初始化
public:
    Test(/* Test* this,*/ int a)
    {
        //书上说 this 作为函数的隐藏参数，传递给函数，每个成员都有，当然他的实现方法我们不用研究，知道每个成员函数都可以使用即可
        this->a = a; //用处 1：用来区分类成员与局部变量
    }
    void Print()
    {
        cout << a << ' ' << this << endl; //用处 2：需要的使用直接用
    }
    Test* GetAddr()
    {
        return this; //用处 3：返回当前对象的地址，外部使用
    }
    Test& GetObject()
    {
        return *this; //用处 4：this 是对象的地址，*this 是对象本身
    }
};
int main(void)
{
    Test t1(12);
    cout << &t1 << ' ' << t1.GetAddr() << endl;
```

```
    //&t1 是当前对象地址，对比 t1 内 this 指针，一样的
    Test t2 = t1.GetObject(); //当然可以只用 t1，只是说 this 就是个普通指针，没什么特殊性
    //用一个对象给另一个对象初始化
    //cout << t1.this << endl; //错误，this 指针只能在类内使用，不能外边用
    return 0;
}
```

拷贝构造/复制构造

1、也是一个构造函数，与其他构造函数构成重载的关系

2、同一个类的一个对象给另一个对象初始化时，调用拷贝构造

3、有默认形式的拷贝构造函数，同普通构造函数一起存在

函数形式：

```
class Test
{
public:
    Test()
    {
        cout << "普通构造" << endl;
    }
    Test(const Test& t)  //参数为当前类的常量引用类型
    {
        cout << "拷贝构造" << endl;
    }
};
```

调用时机

新建一个对象，用已存在的同类对象对其初始化，调用拷贝构造

各种情况如下演示

```
class Test
{
public:
    Test(int a)
    {
        cout << "普通构造" << endl;
    }
    Test(const Test& t)  //参数为当前类的常量引用类型
    {
        cout << "拷贝构造" << endl;
    }
};
Test Fun(Test s)//自定义函数，返回值与参数都是对象
{
    return s;
}
int main(void)
{
    Test t1(12); //走普通构造
    Test t2(t1); //情况 1：t1 给 t2 初始化，走拷贝构造
    Test t3 = t2;//情况 2：t2 给 t3 初始化，走拷贝构造
    Test t4 = Test(12); //情况 3：具体实现取决于编译器，两种情况
    //1、直接用 1 2 给 t4 初始化，走一次普通构造，本编译器采用此
    //2、Test(12)构建一个临时对象，1 2 给临时对象初始化，走一次普通构造
    //    然后临时对象给 t4 初始化，再走一次拷贝构造

    Test* t5 = new Test(t4);
    //情况 4：t4 给 t5 指向的空间(匿名对象)初始化，走拷贝构造
```

```
    Test t6 = Fun(t1);
    //情况 5：调用是实参给形参初始化，即 Test s = t1; 走一次拷贝构造
    //情况 6：函数返回值，两种情况，具体实现取决于编译器
    //1、Fun 函数返回值直接给 t6 初始化，即 Test t6 = s; 走一次拷贝构造，本编译器采用此
    //2、有的编译器在返回值生成临时对象，将 s 初始化给临时对象，走一次构造函数
    //  然后临时对象初始化给 t6，即 Test t6 = 临时对象; 再走一次拷贝构造
    return 0;
}
```

拷贝函数的功能

逐个复制非静态成员，复制的是成员的值

```
class Test
{
private:
    int a;
    double d;
    char str[10] = {0};
public:
    Test(int a = 1, double d = 3.0, const char str[] = "qwerty")
    {
        this->a = a;
        this->d = d;
        for (int i = 0; str[i] != 0; i++)
            this->str[i] = str[i];
    }
    Test(const Test& t)  //参数为当前类的常量引用类型
    {
        this->a = t.a; //不加 this 就行，我只是加上，帮助大家加深一项
        this->d = t.d;
        for (int i = 0; t.str[i] != 0; i++)
            str[i] = t.str[i]; //数组一定要用循环或者库函数
```

```
        //*this = t;   //用这个代替前边的就行
    }
    void Print()
    {
        cout << a << ' ' << d << ' ' << str << endl;
    }
};

int main(void)
{
    Test t1;
    Test t2(t1);
    t2.Print();
    return 0;
}
```

对比默认的构造函数，默认的构造函数是什么都不干，空着

深、浅拷贝

上面这种单纯的值复制，叫浅拷贝，对应的有深拷贝

深拷贝针对指针成员，并且有申请空间与释放空间的操作，需要专门写处理

以下代码演示浅拷贝遇到的问题

```
class Test
{
private:
    int a;
    int* p;
public:
    Test()
    {
        a = 12;
        p = new int[5] {1, 2, 3, 4, 5};
    }
    Test(const Test& t)
    {
```

```
        *this = t;
        //a = t.a;
        //p = t.p;
    }
    ~Test()
    {
        delete p;  //释放
    }
};
int main(void)
{
    Test t1;    //t1 走无参构造，t1 的 p 成员 new 空间
    Test t2(t1);//t2 走拷贝构造，将 t2.p = t1.p; 两个 p 指向同一块空间
    return 0;  //t1t2 调用析构，都 delete p; 释放两次，异常中断
}
```

解决方法，深拷贝，即将 t2 的指针成员也申请空间，如下

```
class Test
{
private:
    int a;
    int* p;
public:
    Test()
    {
        a = 12;
        p = new int[5] {1, 2, 3, 4, 5};
    }
    Test(const Test& t)
    {
        a = t.a;
        p = new int[10]; //要申请空间
        for (int i = 0; i < 5; i++)
            p[i] = t.p[i];  //数组通过循环赋值
    }
```

```
    ~Test()
    {
        delete p;  //释放
    }
};
int main(void)
{
    Test t1;    //t1 走无参构造函数，t1.p 成员 new 空间
    Test t2(t1);//t2 走拷贝构造函数，将 t2.p 与 t1.p 指向不同空间
    return 0;  //t1t2 调用析构，释放各自空间，正常
}
```

浅拷贝正常不用写，只有涉及到深拷贝了，我们需要手写拷贝构造

传递对象时，使用引用可以避免拷贝构造

```
Test Fun1(Test s)//自定义函数，返回值与参数都是对象
{
    return s;
}
Test& Fun2(Test& s)//自定义函数，返回值与参数都是对象
{
    return s;
}
int main(void)
{
    Test t1;
    Test t2 = Fun1(t1); //这会走两次拷贝构造
    Test& t3 = Fun2(t1);//不走任何构造函数。t3 间接引用 t1

    return 0;  //t1t2 调用析构，都 delete p; 释放两次，异常中断
}
```

重载运算符/运算符重载

```
/*************************************************************************
* 对象作为功能与数据的载体，从逻辑上应该可以进行运算
  比如：对象 A + 对象 C，但是已存在的算数规则不能直接应用在对象上
  因为对象的数据成员是自定义了，存在无数种可能性，没办法统一运算规则
  它不像 3 + 3 这种算数，算数是唯一的，不存在多种情况。
* 所以 C++提供了可以对运算符功能拓展的语法特性，叫运算符重载
  前面学习过函数重载，函数重载就是函数名一样，参数列表不一样
  运算符重载就是针对一个运算符，操作数不一样，出现了对象
* 比如：
    12 + 13      这是默认存在的，数据作为操作数的加法
    对象 + 13    这是需要我们重载加法运算符，以计算这个操作数
    13 + 对象
    对象 + 对象
          这两个加法 + ，即是重载的关系，所以叫运算符重载
* 所以，运算符重载要求其中一个操作数必须是对象
*************************************************************************/
```

案例演示：实现两个数组相加，类内重载运算符

```
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
```

```
    /*****************************************************************
    * 类内运算符重载形式 ： 对象 + 数组
    * 函数名：关键字 operator   后接一个运算符+
    * 参数：类内重载，左操作数默认是当前类的对象，参数是右操作数
    * 返回值：返回一个装加后数组的新的对象，可以实现累加
              也可以是 void，但是不能累加了
    * 注意点：此函数是自定义，所以其内代码是任意的，但是不能违背常理
    *****************************************************************/
    Arr operator+(int arr[])  //参数是一个数组，实现：对象+数组
    {
        int tmp[5]; //两个数组相加结果
        for (int i = 0; i < 5; i++)
            tmp[i] = a[i] + arr[i];
        return Arr(tmp);  //返回一个新的 Arr 对象
    }
    /*****************************************************************
    * 类内运算符重载形式 ： 对象 + 对象
    *****************************************************************/
    Arr operator+(const Arr& tp)  //参数是一个对象，实现：对象+对象
    {
        int tmp[5]; //两个数组相加结果
        for (int i = 0; i < 5; i++)
            tmp[i] = a[i] + tp.a[i];
        return Arr(tmp);  //返回一个新的 Arr 对象
    }
    void Print()
    {
        for (int i = 0; i < 5; i++)
            cout << a[i] << ' ';
        cout << endl;
    }
};
```

```
int main(void)
{
    /*****************************************************************
    * 实现两个数组的加法运算，即 a + c，暂定规则是各元素对应相加
      结果是：2 4 6 8 10   规则自定
    *****************************************************************/
    int a[5] = { 1,2,3,4,5 };
    int c[5] = { 1,2,3,4,5 };
    //a + c;  //这语法直接报错
    /*****************************************************************
    * 没有 a+c 这种数组相加的加法，所以我们要重载+法
    * 重载运算符必须至少一个操作数是对象，所以我们创建一个数组类
    * 实现第一种：对象 + 数组，左操作数是对象
    *****************************************************************/
    Arr aa(a); //定义对象初始化，走构造函数
    Arr tp = aa + c; //对象 + 数组
    //此处加法结果返回值是 Arr，初始化给 tp，走默认拷贝构造
    tp.Print();  //tmp 对象装着加完后的数组
    /*****************************************************************
    * 实现第二种：对象 + 对象，左操作数是对象
    *****************************************************************/
    Arr cc(c);
    Arr td = aa + cc; //对象 + 对象
    td.Print();
    /*****************************************************************
    * 实现第三种：数组 + 对象
      此时需要类外重载，因为类内重载左操作数默认是对象
    *****************************************************************/

    return 0;
}
```

案例演示：实现两个数组相加，类外重载运算符

```
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    /*****************************************************************
    * friend 友缘关键字
    * 友缘即朋友，此函数即可使用类内私有成员
    *****************************************************************/
    friend Arr operator+(int c[5], const Arr& tp);
    friend Arr operator+(const Arr& t1, const Arr& t2);
    friend Arr operator+(const Arr& tp, int c[5]);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
    void Print()
    {
        for (int i = 0; i < 5; i++)
            cout << a[i] << ' ';
        cout << endl;
    }
};
/*****************************************************************
* 类外运算符重载形式 ： 数组 + 对象
* 函数名：关键字 operator  后接一个运算符+
* 参数：类外重载，左操作数即左参数，右操作数即右参数
* 返回值：返回一个装加后数组的新的对象，可以实现累加
        也可以是 void，但是不能累加了
* 注意点：此函数是自定义，所以其内代码是任意的，但是不能违背常理
* 由于此函数需要操作类内的私有成员，所以将此函数声明为类的友缘函数
*****************************************************************/
```

```
Arr operator+(int c[5], const Arr& tp)  //两个参数，数组+对象
{
    int tmp[5]; //两个数组相加结果
    for (int i = 0; i < 5; i++)
        tmp[i] = c[i] + tp.a[i];
    return Arr(tmp);  //返回一个新的 Arr 对象
}
Arr operator+(const Arr& tp, int c[5])  //对象+数组
{
    int tmp[5]; //两个数组相加结果
    for (int i = 0; i < 5; i++)
        tmp[i] = c[i] + tp.a[i];
    return Arr(tmp);  //返回一个新的 Arr 对象
}
Arr operator+(const Arr& t1, const Arr& t2) //对象+对象
{
    int tmp[5]; //两个数组相加结果
    for (int i = 0; i < 5; i++)
        tmp[i] = t1.a[i] + t2.a[i];
    return Arr(tmp);  //返回一个新的 Arr 对象
}
int main(void)
{
    /*****************************************************************
    * 实现两个数组的加法运算，即 a + c，暂定规则是各元素对应相加
      结果是：2 4 6 8 10   规则自定
     *****************************************************************/
    int a[5] = { 1,2,3,4,5 };
    int c[5] = { 1,2,3,4,5 };
    Arr aa(a); //定义对象初始化，走构造函数
    /*****************************************************************
    * 实现第三种：数组 + 对象
      此时需要类外重载，因为类内重载左操作数默认是对象
    *****************************************************************/
```

```
    Arr tr = c + aa; //数组 + 对象
    td.Print();
    return 0;
}
/*****************************************************************
* 系统会优先在类内找重载运算符，所以能类内的最好类内重载
  毕竟该运算符运算的是这个对象自己，别的类用不到这个加号
  每个类都需要重载自己的加法，所以定义在类内最符合面向对象编程思想
* 当然，左操作数非本类对象，那必须得类外重载了
*****************************************************************/

/** 输入输出运算符重载 重要，代替 Print**/
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    /*****************************************************************
    * friend 友缘关键字
    * 友缘即朋友，此函数即可使用类内私有成员
    *****************************************************************/
    friend ostream& operator<<(ostream& os, const Arr& tp);
    friend istream& operator>>(istream& in, Arr& tp);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
    void Print()  //重载了<< 就不需要这个 print 了
    {
        for (int i = 0; i < 5; i++)
            cout << a[i] << ' ';
        cout << endl;
    }
```

```
};
/*************************************************************************
* 参数：类外重载，左操作数即左参数，右操作数即右参数
    ostream 是 cout 的类型，cout << 对象，左操作数是 ostream，右是对象
    istream 是 cin 的类型，cin >> 对象，左操作数是 istream，右是对象
* 返回值：返回 ostream 引用，可以实现 cout << 对 1 << 对 2 << 对 3 << endl;
    也可以是 void，但是只能 cout << 对 1;   endl 都加不了
* 注意点：此函数是自定义，所以其内代码是任意的，但是不能违背常理
* 由于此函数需要操作类内的私有成员，所以将此函数声明为类的友缘函数
*************************************************************************/
ostream& operator<<(ostream& os, const Arr& tp)
{
    for (int i = 0; i < 5; i++)
        os << tp.a[i] << ' ';
    return os;  //返回一个 os 对象,可以连续输出
}
istream& operator>>(istream& in, Arr& tp)
{
    for (int i = 0; i < 5; i++)
        in >> tp.a[i] ;
    return in;  //返回一个 is 对象,可以连续输入
}
int main(void)
{
    int a[5] = { 1,2,3,4,5 };
    Arr aa(a);          //定义对象初始化，走构造函数
    cout << aa << endl; //走输出重载，功能跟 Print 函数一样，但是更好用
    cin >> aa;          //走输入重载
    cout << aa << endl;
    return 0;
}
```

重载限制(C++ Primer Plus(第六版) 387 页)

1、不要改变运算符本意，比如减法(-)重载为计算两个对象的和(+)，当然可以这么做，但是不建议这么做。

2、不能违背语句法则，比如除法(%)是两个操作数，不能重载成 1 个操作数的，当然也做不了

3、不能创造新的运算符，重载是对已有的运算符

4、以下运算符不能重载

| | | | | |
|---|---|---|---|---|
| sizeof | . | .* | :: | ?: |
| const_cast | dynamic_cast | reinterpret_cast | static_cast | typeid |

typeid：类型识别的，通过变量或者数据，得到其类型

5、以下运算符只能类内重载

| | | | |
|---|---|---|---|
| = | ( ) | [] | -> |

6、可重载的运算符，大多数类内类外都可以，类内类外取决于操作数顺序

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| + | - | * | / | % | *= | += | -= | /= | %= | |
| ^= | &= | \|= | << | >> | <<= | >>= | & | \| | ^ | ~ |
| ! | && | \|\| | | | | | | | | |
| != | == | > | < | >= | <= | | | | | |
| () | [] | = | -> | ->* | , | | | | | |
| new | delete | new[] | delete[] | | | | | | | |
| ++ | -- | | | | | | | | | |

```
/** 重 载 赋值函数(= )：只能类内，类外直接报错 **/
/*****************************************************************
* 对象 = 数组
* 对象 = 对象：有默认的赋值函数，单纯的所有成员对应直接赋值
      同拷贝构造相似，涉及申请空间的指针成员时，会导致被赋值的对象内存丢失
      两个对象的指针指向同一块空间，析构释放时导致异常中断
      所以存在深赋值的操作，如果这个情况，则不需要重载对象 = 对象
*****************************************************************/
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    int* p;   //指针成员，构造函数 new 空间，析构 delete 空间
    friend ostream& operator<<(ostream& os, const Arr& tp);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
        p = new int[10];
    }
    ~Arr()
    {
        delete p;
    }
    //默认左操作数是对象：对象 = 数组
    //返回值可以实现连续赋值
    Arr& operator=(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
        return *this;
    }
```

```
    //对象=对象，浅赋值，默认存在函数
    /*Arr& operator=(const Arr& tp)
    {
        for (int i = 0; i < 5; i++)
            a[i] = tp.a[i];
        p = tp.p;  //使得两个 p 指向同一块空间
        return *this;
    }*/
    //深浅不能同时存在
    Arr& operator=(const Arr& tp)
    {
        for (int i = 0; i < 5; i++)
            a[i] = tp.a[i];
        for (int i = 0; i < 10; i++)
            p[i] = tp.p[i];  //将 tp.p 空间的值赋值给 p
        return *this;
    }
};
ostream& operator<<(ostream& os, const Arr& tp)
{
    for (int i = 0; i < 5; i++)
        os << tp.a[i] << ' ';
    return os;  //返回一个 os 对象,可以连续输出
}
int main(void)
{
    int a[5] = { 1,2,3,4,5 };
    int c[5] = { 6,7,8,9,0 };
    Arr aa(a), cc(c);           //定义对象初始化，走构造函数
    aa = cc = a;
    cout << cc << endl;
    cout << aa << endl;
    return 0;
}
```

```
/** 函数调用运算符()  **/
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    friend ostream& operator<<(ostream& os, const Arr& tp);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
    /** 函数调用运算符重载 **/
    /************************************************
    * 内可以是输出对象成员的数据，同 << 或者 Print
    * 内可以是输出对象的成员布局，如下
    * 返回值可有可无
    ************************************************/
    void operator()()
    {
        cout << "int a[5];" << endl;
        cout << "Arr(int arr[])" << endl;
        cout << "void operator()()" << endl;
        cout << "friend ostream& operator<<(ostream& os, const Arr& tp)" << endl;
    }
    /** 函数调用运算符重载：带参数 **/
    /************************************************
    * 参数可以是对成员重新赋值
    * 参数个数，类型随意，功能随意
    * 返回值随意,带返回值 Arr&,就可以直接 cout <<
    ************************************************/
    Arr& operator()(int aa[], double d)
    {
        for (int i = 0; i < 5; i++)
```

```
            a[i] = aa[i];
        return *this;
    }
};
ostream& operator<<(ostream& os, const Arr& tp)
{
    for (int i = 0; i < 5; i++)
        os << tp.a[i] << ' ';
    return os;  //返回一个 os 对象,可以连续输出
}
int main(void)
{
    int a[5] = { 1,2,3,4,5 };
    int c[5] = { 6,7,8,9,0 };
    Arr aa(a);   //定义对象初始化，走构造函数
    aa();        //函数重载运算符，对象调用
    //带参数的函数调用运算符重载，有返回值 Arr&，便可以直接输出
    cout << aa(c, 5.3) << endl;
    //如返回值为 void,则如下
    aa(c, 5.3);          //单独调用
    cout << aa << endl; //再输出
    return 0;
}
```

```
/** 下标运算符重载[]：只能类内 **/
/************************************************
* 下标运算符一般用于数组式连续空间的访问
************************************************/
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    friend ostream& operator<<(ostream& os, const Arr& tp);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
    /************************************************************
    * 使用形式：对象[下标]
    * 所以有一个整数参数
    * 返回值为 int&，数组时，a[n]是可读可写，可取地址的
    ************************************************************/
    int& operator[](int n)
    {
        return a[n];
    }
};
ostream& operator<<(ostream& os, const Arr& tp)
{
    for (int i = 0; i < 5; i++)
        os << tp.a[i] << ' ';
    return os;  //返回一个 os 对象,可以连续输出
}
```

```
int main(void)
{
    int a[5] = { 1,2,3,4,5 };
    Arr aa(a);   //定义对象初始化，走构造函数
    cout << aa[1] << endl; //输出
    aa[2] = 34;            //重新赋值
    cout << aa << endl;
    cin >> aa[3];          //指定元素输入
    cout << aa << endl;
    return 0;
}
```

```
指针取成员运算符：->
/*****************************************************
* 不能有参数
* 返回值必须是可以使用->的类型，比如类指针，结构体指针
* 调用：对象->成员，不必使用对象指针了
    AA->fun();   相当于 AA 对象换成 this
*****************************************************/
Arr* operator->()
{
    return this;
}
void Fun()
{
    cout << "成员函数 fun" << endl;
}
/*****************************************************
* 指针取成员运算符：->
* 不能有参数，所以不能重载
* 返回值必须是可以使用->的类型，比如类指针，结构体指针
* 调用：对象->成员，不必使用对象指针了
    AA->a; 此时 AA 位置相当于替换成&g，g 调用自己的 a
*****************************************************/
struct Node
{
    int a;
}g = {12};
Node* operator->()
{
    return &g;
}
```

```
/*************************************************************
* 成员指针运算符：->*
* p 叫成员指针变量，类型为：int Arr::* 专门指向 Arr 成员的指针
* &Arr::g 成员地址
* p 必须通过 Arr 对象调用
*************************************************************/
int Arr::*p = &Arr::g;  //公有成员 g,方便调用
int main()
{
    int a[5] = { 1,2,3,4,5 }, c[5] = { -11,-12,13,-14,15 };
    Arr AA(a), CC(c);
    cout << (&AA)->*p << endl;  //重载前的默认调用
    return 0;
}
/*************************************************************
* 类内重载：->*
* 返回值是：成员的引用
* 参数是成员指针所指向的那个成员类型，要调用这个 g 成员
*************************************************************/
int& operator->*(int Arr::* g)
{
    return this->g;
}
/*************************************************************
* 类外重载：->*
* 返回值是：成员的引用
* 参数是成员指针所指向的那个成员类型，要调用这个 g 成员
*************************************************************/
int& operator->*(Arr& tp, int Arr::* g)
{
    return tp.g;
}
int main()
{
```

```
    int a[5] = { 1,2,3,4,5 }, c[5] = { -11,-12,13,-14,15 };
    Arr AA(a), CC(c);
    cout << (&AA)->*p << endl;  //重载前的默认调用
    cout << AA->*p << endl;     //重载后可以如此调用
    return 0;
}



/*************************************************************
* 负号运算符：-
* 类外:一个右操作数,为当前类对象，要声明友缘
* 类内:无参数
* 逻辑采用：将数组的各元素加个负号。正转负，负转正
*************************************************************/
Arr operator-(const Arr& tp)
{
    int tmp[5] = { 0 };
    for (int i = 0; i < 5; i++)
        tmp[i] = -tp.a[i];
    return Arr(tmp);  //返回一个新的对象
}
/*************************************************************
* 负号运算符：+
* 类外:一个右操作数,为当前类对象，要声明友缘
* 类内:无参数
* 逻辑采用：将数组的各元素加个正号。值不变
*************************************************************/
Arr operator+(const Arr& tp)
{
    int tmp[5] = { 0 };
    for (int i = 0; i < 5; i++)
        tmp[i] = +tp.a[i];
    return Arr(tmp);
}
```

```
int main()
{
    int a[5] = { 1,2,3,4,5 },c[5] = { -11,-12,13,-14,15 };
    Arr AA(a), CC(c);
    cout << -AA << endl;
    cout << +CC << endl;
    return 0;
}

/* 类内重载取地址运算符：& */
/* 无参数，如果有参数就成了按位与重载：对象&对象 */
Arr* operator&()
{
    return this; //也可以返回某成员的地址
}
/* 类外重载取地址运算符：& */
/* 1 个参数，&对象 */
/* 类外不能使用 this，此时也不能返回&tp，会无限递归 */
/* 所以，可以返回指定成员的地址，如下，需要加友缘*/
const int* operator&(const Arr& tp)
{
    return &tp.a[0]; //也可以返回某成员的地址
}
int main()
{
    int a[5] = { 1,2,3,4,5 };
    int c[5] = { -11,-12,13,-14,15 };
    Arr AA(a), CC(c);
    cout << &AA;   //调用形式
    return 0;
}
```

```
/* 类内重载按位取反运算符：~ */
/* 无参数 */
Arr operator~()
{
    int tmp[5] = { 0 };
    for (int i = 0; i < 5; i++)
        tmp[i] = ~a[i];
    return Arr(tmp); //也可以返回某成员的地址
}

/* 类外重载按位取反运算符：~ */
/* 1 个参数，~对象 */
/* 需要加友缘*/
Arr operator~(const Arr& tp)
{
    int tmp[5] = { 0 };
    for (int i = 0; i < 5; i++)
        tmp[i] = ~tp.a[i];
    return Arr(tmp); //也可以返回某成员的地址
}
int main()
{
    int a[5] = { 1,2,3,4,5 };
    int c[5] = { -11,-12,13,-14,15 };
    Arr AA(a), CC(c);
    cout << ~AA;   //调用形式
    return 0;
}
```

```
/** 自加运算符：++ **/
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    friend ostream& operator<<(ostream& os, const Arr& tp);
    friend Arr& operator++(Arr& tp);
    friend Arr operator++(Arr& tp, int n);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
    /** 类内重载:前置++                          **/
    /** 无参数                                   **/
    /** 返回值为当前对象的引用，可以直接输出    **/
    Arr& operator++()
    {
        for (int i = 0; i < 5; i++)
            a[i] += 1;  //所有元素自加 1
        return *this;
    }
    /** 类内重载:后置++                          **/
    /** 带一个 int 参数，占位的，无使用          **/
    /** 返回值自加前的值的对象，可以直接输出 **/
    Arr operator++(int n)
    {
        int tmp[5] = { 0 };
        for (int i = 0; i < 5; i++)
        {
            tmp[i] = a[i];
            a[i] += 1;      //所有元素自加 1
        }
```

```
        return Arr(tmp);
    }
};

/** 类外重载:前置++                          **/
/** ++对象，一个类对象                        **/
/** 返回值参数这个对象，可以直接输出          **/
/** 声明友缘                                  **/
Arr& operator++(Arr& tp)
{
    for (int i = 0; i < 5; i++)
        tp.a[i] += 1;  //所有元素自加 1
    return tp;
}
/** 类外重载:后置++                                    **/
/** 两个参数，一个对象，int 参数，占位的，无使用        **/
/** 返回值自加前的值的对象，可以直接输出               **/
/** 声明友缘                                           **/
Arr operator++(Arr& tp, int n)
{
    int tmp[5] = { 0 };
    for (int i = 0; i < 5; i++)
    {
        tmp[i] = tp.a[i];
        tp.a[i] += 1;       //所有元素自加 1
    }
    return Arr(tmp);
}
ostream& operator<<(ostream& os, const Arr& tp)
{
    for (int i = 0; i < 5; i++)
        os << tp.a[i] << ' ';
    return os;  //返回一个 os 对象,可以连续输出
}
```

```
int main(void)
{
    int a[5] = { 1,2,3,4,5 };
    Arr aa(a);
    return 0;
}

/** 自减运算符：-- **/
class Arr
{
private:
    int a[5]; //固定 5 个元素，不写过于复杂的
    friend ostream& operator<<(ostream& os, const Arr& tp);
    friend Arr& operator--(Arr& tp);
    friend Arr operator--(Arr& tp, int n);
public:
    Arr(int arr[])
    {
        for (int i = 0; i < 5; i++)
            a[i] = arr[i];
    }
    /** 类内重载:前置++                          **/
    /** 无参数                                   **/
    /** 返回值为当前对象的引用，可以直接输出 **/
    Arr& operator--()
    {
        for (int i = 0; i < 5; i++)
            a[i] -= 1;  //所有元素自加 1
        return *this;
    }
    /** 类内重载:后置++                          **/
    /** 带一个 int 参数，占位的，无使用          **/
    /** 返回值自加前的值的对象，可以直接输出 **/
    Arr operator--(int n)
```

```
    {
        int tmp[5] = { 0 };
        for (int i = 0; i < 5; i++)
        {
            tmp[i] = a[i];
            a[i] -= 1;      //所有元素自加 1
        }
        return Arr(tmp);
    }
};

/** 类外重载:前置--                          **/
/** ++对象，一个类对象                        **/
/** 返回值参数这个对象，可以直接输出          **/
/** 声明友缘                                  **/
Arr& operator--(Arr& tp)
{
    for (int i = 0; i < 5; i++)
        tp.a[i] -= 1;  //所有元素自加 1
    return tp;
}
/** 类外重载:后置--                                      **/
/** 两个参数，一个对象，int 参数，占位的，无使用          **/
/** 返回值自加前的值的对象，可以直接输出                  **/
/** 声明友缘                                              **/
Arr operator--(Arr& tp, int n)
{
    int tmp[5] = { 0 };
    for (int i = 0; i < 5; i++)
    {
        tmp[i] = tp.a[i];
        tp.a[i] -= 1;      //所有元素自加 1
    }
    return Arr(tmp);
```

```
}
ostream& operator<<(ostream& os, const Arr& tp)
{
    for (int i = 0; i < 5; i++)
        os << tp.a[i] << ' ';
    return os;  //返回一个 os 对象,可以连续输出
}
int main(void)
{
    int a[5] = { 1,2,3,4,5 };
    Arr aa(a);
    return 0;
}
```

申请释放的重载，一般是类内重载，因为类外重载会影响到正常已存在的 new，delete 功能，类内外重载的形式参数都一样

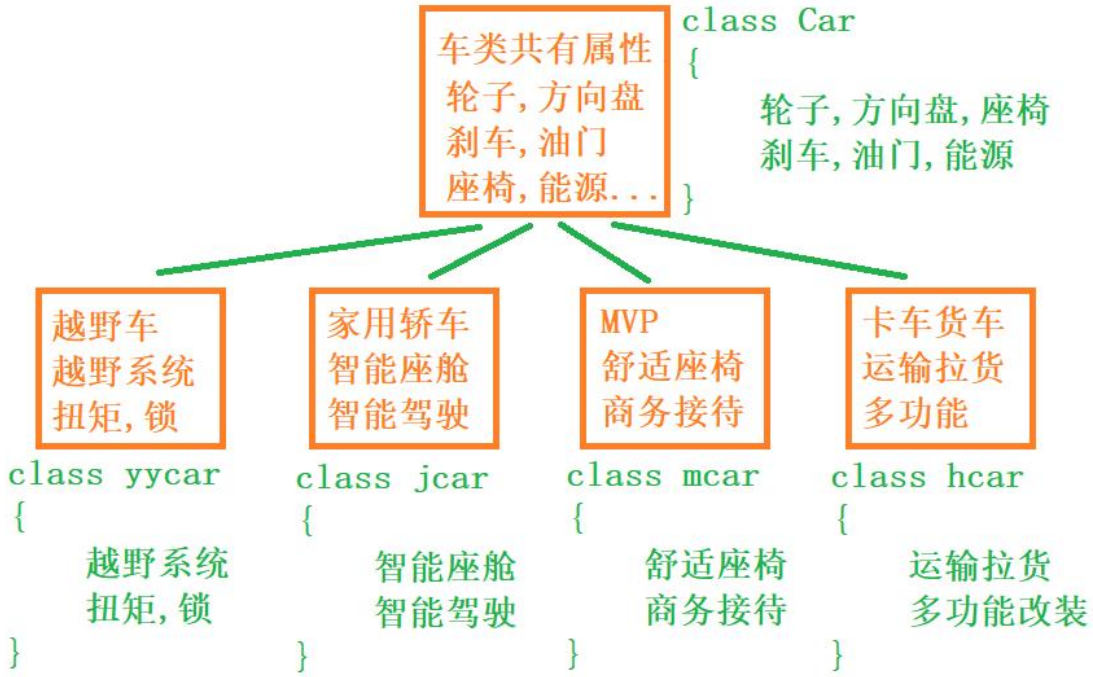
```
#include <iostream>
#include <cstdlib>
using namespace std;

class Arr
{
public:
    //一个参数，即字节数，返回值为空间首地址 void*
    void* operator new(size_t size)
    {
        cout << "nihao" << size << endl; //内部可以做一些专属的操作
        //return ::operator new(size);   //调用全局版的 new 运算符
        return malloc(size);             //调用 malloc  头文件 cstdlib
malloc
    }
    //一个参数，即字节数，返回值为空间首地址 void*
    void* operator new[](size_t size)
    {
        cout << "buhao" << size << endl;  //内部可以做一些专属的操作
        return ::operator new(size);      //调用全局版的 new
    }
    //一个参数，即空间首地址，返回值为空间首地址 void*
    void operator delete(void* p)
    {
        cout << "world" << endl;
        ::operator delete(p);
        //free(p);
    }
    void operator delete[](void* p)
    {
        cout << "shijie" << endl;
        ::operator delete[](p);
```

```
    }
};

int main(void)
{
    Arr* p = new Arr();   //走 new
    Arr* q = new Arr[5];  //走 new[]
    delete p;             //走 delete
    delete[] q;           //走 delete[]

    return 0;
}
```

继承：功能分析


功能就是类的数据成员与函数成员
四种车类前三行成员是一模一样的，后两行是各自的特有属性
从代码角度讲，四个类的前三行成员的代码是重复的，那C++的继承功能就是让这些重复的代码只写一次

共同的属性只写一次，是个什么关系？如下：




底下四个小类也具备公共类的属性，让小类拥有公共类的属性的语法，叫继承

所以：继承提升了代码的复用性，重用性

```
/*********************************************************************
* 继承的作用:
* 1.访问控制，通过 protected 等关键字管理成员的访问权限
* 2.代码复用，减少重复编写相同的代码，提升工作效率
* 3.实现多态，允许子类重写父类方法，增强程序的灵活性
* 4.使得代码的拓展与修改更加的方便，更具想象力
* 5.定义接口规范，通过抽象基类强制子类实现特定方法
* 6.支持一些设计模式的实现
*********************************************************************/
公共类：基类，平时口头也叫父类
各自类：派生类，平时口头也叫子类
/** 一段继承代码 **/
class Car  //基类
{
public:
    void CheLun()
    {
        cout << "我有轮子" << endl;
    }
    void FangXiang()
    {
        cout << "我有方向盘" << endl;
    }
};
class YueYeCar : public Car  //公有继承 Car,冒号
{
public:
    void YueYeSys()
    {
        cout << "我有越野控制系统" << endl;
    }
};
class KaCar : public Car    //公有继承 Car,冒号
{
```

```
public:
    void GaiZhuangSys()
    {
        cout << "我可以改装成多功能工具车" << endl;
    }
};
/*********************************************************
* 基类:   Car
* 派生类：YueYeCar  KaCar
*********************************************************/
int main(void)
{
    //Car 类对象,只能调用 Car 的成员
    Car car;
    car.CheLun();
    car.FangXiang();
    //YueYeCar 类对象，可以调用 Car 类成员，可以调用 YueYeCar 成员
    YueYeCar yycar;
    yycar.CheLun();
    yycar.FangXiang();
    yycar.YueYeSys();
    //KaCar 类对象，可以调用 Car 类成员，可以调用 KaCar 成员
    KaCar kcar;
    kcar.CheLun();
    kcar.FangXiang();
    kcar.GaiZhuangSys();
    //YueYeCar、KaCar 两个子类没有关系，各是各的
    return 0;
}
```

基类的成员访问权限对派生类的限制

```
class Car  //基类
{
private:
    int a = 1;
protected:
    int c = 2;
public:
    int e = 3;
};
class KaCar : public Car    //公有继承 Car,冒号
{
public:
    void print()
    {
        //报错，基类 peivate 只能在[基类]直接使用
        cout << a << endl;
        //成功，基类 protected 可以在[基类/派生类]直接使用
        cout << c << endl;
        //成功，基类 public 可以在[基类/派生类/类外]直接使用
        cout << e << endl;
    }
};
int main(void)
{
    KaCar kc;
    //报错，基类 peivate 只能在[基类]直接使用
    cout << kc.a << endl;
    //报错，基类 protected 可以在[基类/派生类]直接使用
    cout << kc.c << endl;
    //成功，基类 public 可以在[基类/派生类/类外]直接使用
    cout << kc.e << endl;
    return 0;
}
```

继承的权限功能

```
class KaCar : private Car
/*************************************************************************
* 私有继承:private
* 1.继承后，基类 public,protected 成员，相当于派生类的 private 成员
      基类的 private 权限不变，还是只能在基类内使用
* 2.不可以通过派生类的对象，访问基类 public 成员
* 3.派生类的子类不可以访问，基类 public,protected 成员
*************************************************************************/

class KaCar : protected Car
/*************************************************************************
* 保护继承:protected
* 1.继承后，基类 protected,public 相当于成为了派生类的 protected 成员
      基类的 private 权限不变，还是只能在基类内使用
* 2.不可以通过派生类的对象，访问基类 public 成员
* 3.派生类的子类可以访问，基类 public,protected 成员
*************************************************************************/

class KaCar : public Car
/*************************************************************************
* 公有继承:public
* 1.功能：继承后，一切照常
*************************************************************************
***/
```

基类子类成员名相同

```
class CA          //基类
{
public:
    int e = 3;
};
class CC : public CA
{
public:
    int e = 4;  //子类成员与父类重名，子类内名字覆盖父类
    void Fun()
    {
        cout << e << endl;      //使用的是派生类 CC 的 e
        cout << CA::e << endl; //使用的是基类 CA 的 e
    }
};
int main(void)
{
    CC c;  //定义派生类对象
    c.Fun();
    cout << c.e << endl;      //使用的是派生类 CC 的 e
    cout << c.CA::e << endl; //使用的是基类 CA 的 e
    return 0;
}
```

```
/**********************************
* 构造函数调用顺序：由父到子
* 析构函数调用顺序：由子到父
**********************************/
class CA
{
public:
    CA()  { cout << "爷爷" << endl;  }
    ~CA() { cout << "~爷爷" << endl; }
};
class CC : public CA
{
public:
    CC()  { cout << "父亲" << endl;  }
    ~CC() { cout << "~父亲" << endl; }
};
class CD : public CC
{
public:
    CD()  { cout << "儿子" << endl;  }
    ~CD() { cout << "~儿子" << endl; }
};
int main(void)
{
    CD d; //定义儿子对象
    return 0;
}
/**************************************
* 构造：爷爷   父亲   儿子
* 析构：~儿子  ~父亲  ~爷爷
**************************************/
```

单继承中，基类构造有参数，通过初始化列表传递

```
class CA
{
public:
    CA(int a)   { cout << "CA" << endl;  }
    ~CA()       { cout << "~CA" << endl; }
};
class CC : public CA
{
public:
    CC(int a) : CA(a)   { cout << "CC" << endl;  }
    ~CC()               { cout << "~CC" << endl; }
};
```

多继承中，多个基类构造有参数，通过初始化列表传递

```
class CA
{
public:
    CA(int a)   { cout << "CA" << endl;  }
    ~CA()       { cout << "~CA" << endl; }
};
class CC
{
public:
    CC(int a) { cout << "CC" << endl;  }
    ~CC()     { cout << "~CC" << endl; }
};
class CD : public CC, CA    //多继承，顺序影响基类构造析构的顺序
{
public:
    CD(int a) : CA(a), CC(a)    { cout << "CD" << endl;  }
    ~CD()                       { cout << "~CD" << endl; }
};
```

多个基类的构造顺序与析构顺序，由继承的顺序决定，即先调用 CC 的构造与析构，后调用 CA 的

基类的指针(引用)可以指向派生类类型的地址

一般来说，指针(引用)只能指向(引用)同类型的对象，继承这里就不遵循这个

```
class CA
{
public:
    void funa() { cout << "CA::funa()" << endl; }
};
class CC : public CA
{
public:
    void func() { cout << "CC::func()" << endl; }
};
int main(void)
{
    CA* pa = new CC; //基类指针指向子类空间
    CC c;
    CA& pp = c;       //基类引用子类对象

    //但是，pa,pp 只能调用 CA 类的成员，不能调用子类的成员
    pa->funa(); //正确 CA::funa()
    pp.funa();  //正确 CA::funa()

    pa->func(); //错误
    pp.func();  //错误

    //子类对象是可以调用基类的 public 成员
    c.funa();  //调用基类 public 成员 CA::funa()
    c.func();  //调用派生类成员       CC::func()
    delete pa;
    return 0;
}
```

如何让父类指针调用子类的函数呢？ 虚函数

虚函数的语法形式

1、派生类重写基类函数，即函数头一样

2、基类函数定义成虚函数，virtual

```
class CA
{
public:
    virtual void fun() { cout << "CA::funa()" << endl; }
};
class CC : public CA
{
public:
    void fun() { cout << "CC::func()" << endl; return 0; }
    //派生类的 fun，函数头必须与父类的虚函数一模一样的，才能达成
};
int main(void)
{
    CA* pa = new CC; //基类指针指向子类空间
    CC c;
    CA& pp = c;       //基类引用子类对象
    //但是，pa,pp 能调用子类的成员 fun
    pa->fun(); //调用子类的 fun  CC::func()
    pp.fun();  //调用子类的 fun  CC::func()

    delete pa;
    return 0;
}
```

所以，虚函数的作用是使得父类的指针或者引用，指向子类的对象，便可以调用子类重写父类虚函数的成员

虚函数与多态

多态：一种写法，多种执行状态。函数重载，默认参数，运算符重载

虚函数的多态：

```
class CA
{
public:
    virtual void fun() { cout << "CA::funa()" << endl; }
};
class CC : public CA
{
public:
    void fun() { cout << "CC::func()" << endl; }
};
class CD : public CA
{
public:
    void fun() { cout << "CD::func()" << endl; return 0; }
};
int main(void)
{
    CA* pa = new CC; //基类指针指向子类空间
    //CA* pa = new CD;
    pa->fun();
    //new CC 执行结果是 CC 的 fun
    //new CD 执行结果是 CD 的 fun
    //一种调用，多种执行状态
    delete pa;
    return 0;
}
```

动态联编，静态联编

动态联编：程序在运行时确定调用执行的目标

静态联编：程序在编译时确定调用执行的目标

```
class CA
{
public:
    virtual void fun() { cout << "CA::funa()" << endl; }
    void fun1() { cout << "CA::fun1()" << endl; }
};
class CC : public CA
{
public:
    void fun() { cout << "CC::func()" << endl; }
};
class CD : public CA
{
public:
    void fun() { cout << "CD::func()" << endl; return 0; }
};
int main(void)
{
    CA* pa = new CC; //基类指针指向子类空间
    //CA* pa = new CD;
    pa->fun();   //动态联编，运行时确定执行代码
    pa->fun1();  //静态联编，编译时确定执行代码
    delete pa;
    return 0;
}
```

虚析构

多态状态下：构造函数调用顺序：先基类，后派生类

析构函数调用顺序：只调用基类的

解决办法：将基类的析构函数声明成虚函数

注意：构造函数不能是虚函数

```
class CA
{
public:
    CA()  { cout << "CA" << endl; }
    virtual ~CA() { cout << "~CA" << endl; }
};
class CC : public CA
{
public:
    CC()  { cout << "CC" << endl; }
    ~CC() { cout << "~CC" << endl; }
};
int main(void)
{
    CA* pa = new CC;
    delete pa;
    return 0;
}
```

多态的实现原理：虚表  虚函数列表

多态的具体实现取决于编译器的作者，虚表的实现方式，是其中一种

```
/*************************************************************************
* 虚表：
* 虚表是一个元素为虚函数地址的数组，每个元素都是一个虚函数地址
* 虚表不是成员，他的首地址装对象首部
* 实现过程：
* 1、带虚函数的对象，其对象空间的首(4/8)字节装着虚表的地址
* 2、创建子类对象，会先构造父类的空间，将父类的虚函数依次装入虚表
* 3、再构造子类空间，将子类的虚函数依次装入虚表
* 4、如果有重写父类的虚函数，则在表中替换覆盖掉父类的虚函数地址
* 调用过程：
* 1、父类指针调用的是虚函数
* 2、则在表中找到该函数
* 3、执行该函数
*************************************************************************/
class CA
{
public:
    virtual void fun1() { cout << "基类:fun1" << endl; }
    virtual void fun2() { cout << "基类:fun2" << endl; }
    virtual void fun3() { cout << "基类:fun3" << endl; }
    virtual void fun4() { cout << "基类:fun4" << endl; }
    void fun7() { cout << "基类:fun4" << endl; }
};
class CC : public CA
{
public:
    virtual void fun2() { cout << "派生类:fun2" << endl; }
    virtual void fun3() { cout << "派生类:fun3" << endl; }
    virtual void fun5() { cout << "派生类:fun5" << endl; }
    void fun6() { cout << "派生类:fun6" << endl; }
};
CA* pa = new CC;  //定义对象
```

先构造父类空间，表 1

| | | | |
|---|---|---|---|
| fun1(基) | fun2(基) | fun3(基) | fun4(基) |

在构造子类空间，在表 1 的基础上变化如下，表 2

| | | | | |
|---|---|---|---|---|
| fun1(基) | fun2(派) | fun3(派) | fun4(基) | fun5(派) |

此时的虚表即为表 2，(只有一张表 2，俩表展示的是表的变化过程)
非虚函数不入表
pa->虚函数，就是根据名字，到表里找，调用表中的实时函数

```
/*****************************************************************
* 手抓虚表
* 虚表是一个指针数组，由于该数组成员是不固定的函数地址类型，所以，
  无法像整型数组那样直接取，但是，可确定一点，数组元素为 4/8 字节大小
* 步骤
* 1、取数组首地址，数组名字，因为要取 8 字节，所以用 long long 取
  CA* pa = new CC;
  long long addr = ((long long*)pa)[0];//x64 环境，强取前 8 字节
  //int addr = ((int*)pa)[0];          //x86 环境，强取前 4 字节

* 2、数组地址转换
  addr 变量里装的是数组地址，要把该地址转成 long long*型，可一次取 8 个
  long long* faddr = (long long*)addr;

* 3、取数组元素地址，数组元素就是虚函数地址
  faddr + 0;
  faddr + 1;
  faddr + 2;  //依次为各元素地址

* 4、取数组元素，数组元素就是各函数地址
  *(faddr + 0),*(faddr + 1),*(faddr + 2),*(faddr + 3),
  即 faddr[0],faddr[1],faddr[2],faddr[3],
```

```
* 5、将地址换成相应的虚函数类型，函数什么类型转成什么类型
  void (*f1)() = (void (*)())faddr[0];
  void (*f2)() = (void (*)())faddr[1];
  void (*f3)() = (void (*)())faddr[2];
  void (*f4)() = (void (*)())faddr[3];

* 6、函数调用
  f1();
  f2();
  f3();
  f4();
**********************************************************************/
```

```
string类，专门处理字符串的
#include <iostream>
#include <string>
//C语言里是string.h
using namespace std;
//string的名字空间也是std
/** 构造函数 ************************************************************
* string();
* 字符参数
* string(size_type count, value_type ch);
* 字符串参数
* string(const value_type* s);
* string(const value_type* s, size_type count);
* string(const value_type* s, size_type pos, size_type count);
* string对象参数
* string(const string& other);
* string(const string& other, size_type pos);
* string(const string& other, size_type pos, size_type count);
* string(string&& other);//(string)"erty" string("qwe")
**********************************************************************/

/***** 赋值类函数 ******************************************************
* append  string对象结尾增加字符
* 字符参数
* string& append(size_type count, value_type ch);
* 字符串参数
* string& append(const value_type* s);
* string& append(const value_type* s, size_type count);
* string& append(const value_type* s, size_type pos, size_type count);
* string对象参数
* string& append(const string& str);
* string& append(const string& str,
                 size_type pos, size_type count = npos);
**********************************************************************
```

```
* assign  string 对象重新赋值
* 字符参数
* string& assign(size_type count, char ch);
* 字符串参数
* string& assign(const value_type* s);
* string& assign(const value_type* s, size_type count);
* string& assign(const value_type* s, size_type pos, size_type count);
* string 对象参数
* string& assign(const string& str);
* string& assign(const string& str,
                 size_type pos, size_type count = npos);
**************************************************************************
* insert  在指定位置接入一个元素/一组元素/或者一个 string 对象
* 字符参数
* string& insert(size_type index, size_type count, CharT ch);
* 字符串参数
* string& insert(size_type index, const CharT * s);
* string& insert(size_type index, const CharT * s, size_type count);
* string& insert(size_type index,
                 const value_type* s, size_type pos, size_type count);
* string 对象参数
* string& insert(size_type index, const string& str);
* string& insert(size_type index,
          const string& str, size_type s_index, size_type count = npos);
**************************************************************************
* replace  用指定的字符或 string 段或字符串段替换指定位置的元素
* 字符参数
* string& replace(size_type pos, size_type count,
                  size_type count2, CharT ch);
* 字符串参数
* string& replace(size_type pos, size_type count,
                  const value_type* cstr);
* string& replace(size_type pos, size_type count,
                  const value_type* cstr, size_type count2);
```

```
* string& replace(size_type pos, size_type count,
                const value_type* cstr, size_type pos, size_type count);
* string 对象参数
* string& replace(size_type pos, size_type count, const string& str);*
string& replace(size_type pos, size_type count,
          const string& str, size_type pos2, size_type count2 = npos);
**************************************************************************
* copy  指定下标处，拷贝一段装进一个指定的字符数组中
* size_type copy( value_type* dest, size_type count, size_type pos = 0);
**************************************************************************
* clear  清空所有对象字符元素
* void clear();
**************************************************************************
* erase  移除指定下标元素，或者指定范围下标元素
* basic_string& erase(size_type index = 0, size_type count = npos);
**************************************************************************/
```

```
** 元素操作类函数 ************************************************
* at  返回 string 对象指定下标的元素引用
* CharT& at( size_type pos );
* const CharT& at( size_type pos ) const;
*****************************************************************
* c_str  返回对象中字符串的地址，const char*
* const CharT* c_str() const;
*****************************************************************
* data  返回 string 对象内字符串的地址，char*
* const CharT* data() const;
* CharT* data() noexcept;
*****************************************************************
* front  返回对象首元素引用
* CharT& front();
* const CharT& front() const;
*****************************************************************
* back  返回对象尾元素引用
* CharT& back();
* const CharT& back() const;
*****************************************************************
* pop_back  删除 string 对象最后一个元素
* void pop_back();
*****************************************************************
* push_back   string 对象最后添加一个元素
* void push_back( CharT ch );
*****************************************************************
```

```
/** 对象属性类函数 ***************************************************
* empty  判断 string 对象是否是空的,空返回 1，非空返回 0
* bool empty() const;
*********************************************************************
* length/size   返回当前 string 对象中的元素个数
* size_type size() const;
* size_type length() const;
*********************************************************************
* capacity  返回 string 对象的容量,初始 15，每次增加 16，不同环境不一样
* size_type capacity() const;
*********************************************************************
* max_size   返回当前系统可装的最大的字符串的字符数
* size_type max_size() const;
*********************************************************************
* shrink_to_fit  丢弃字符串的多余容量，16*n
* void shrink_to_fit();
*********************************************************************
* reserve   重置容量，设置为至少与指定数字一样大的数字,15+16*n
* constexpr void reserve( size_type new_cap );
*********************************************************************
* resize   重新指定字符数量，根据需要增加/删除元素
* void resize( size_type count );
* void resize( size_type count, CharT ch );
*********************************************************************/
```

```
/**** operator ***************************************************
* +     对象与对象，对象与字符串，对象与字符相加
* +=    string 对象，拼接一段
* string& operator+=( const string& str );
* string& operator+=( CharT ch );
* string& operator+=( const CharT* s );
* []    下标访问，返回元素引用
* CharT& operator[]( size_type pos );
* const CharT& operator[]( size_type pos ) const;
* =     对象重新赋值，赋值对象，赋值字符串，赋值字符
* !=  ==  <  <=  >  >=   关系运算符，结果是 bool
* <<  >>   输入输出运算符重载
*****************************************************************
* getline  输入，非成员
* getline(cin, s);
*****************************************************************
* compare  比较两个 string 对象,也可比较两个对象指定段
* int compare( const string& str ) const;
* int compare( size_type pos1, size_type count1,
              const string& str ) const;
* int compare( size_type pos1, size_type count1,
      const string& str,size_type pos2, size_type count2) const;
* int compare( const CharT* s ) const;
* int compare( size_type pos1, size_type count1,
              const CharT* s ) const;
* int compare( size_type pos1, size_type count1,
              const CharT* s, size_type count2 ) const;
*****************************************************************
* swap  交换两个 string 对象
* void swap( string& other );
*****************************************************************/
```

```
/**** 转换函数,非成员 ***********************************************
* stod 转换成 double
* stof 转换成 float
* stold   转换成 long double
* float       stof ( const string& str);
* double      stod ( const string& str);
* long double stold( const string& str);
* 参数 2 返回字符串的字符位数
*********************************************************************
* stoi    转换成 int
* stol    转换成 long
* stoll   转换成 long long
* int stoi(const std::string& str,
           std::size_t* pos = nullptr, int base = 10 );
* long stol(const std::string& str,
            std::size_t* pos = nullptr, int base = 10 );
* long long stoll(const std::string& str,
                  std::size_t* pos = nullptr, int base = 10 );
* 参数 2：返回 string 中的数字字符的位数
* 参数 3：指示字符串是几进制
*********************************************************************
* stoul    转换成 unsigned long
* stoull   转换成 unsigned long long
* unsigned long stoul(const std::string& str,
                      std::size_t* pos = nullptr, int base = 10 );
* unsigned long long stoull(const std::string& str,
                      std::size_t* pos = nullptr, int base = 10 );
*********************************************************************
* to_string   一个数值转换成 string
* std::string to_string( int value );
* std::string to_string( long value );
* std::string to_string( long long value );
* std::string to_string( unsigned value );
* std::string to_string( unsigned long value );
```

```
* std::string to_string( unsigned long long value );
* std::string to_string( float value );
* std::string to_string( double value );
* std::string to_string( long double value );
*****************************************************************/

/** string 对象子串查找 ***********************************************
* find  从左到右找到第一个目标子串
* size_type find( CharT ch, size_type pos = 0 ) const;
* size_type find( const CharT* s, size_type pos = 0 ) const;
* size_type find( const CharT* s, size_type pos, size_type count) const;
* size_type find( const string& str, size_type pos = 0 ) const;
* 参数 pos:搜索起始下标位置，左到右，参数 count:子串首多少个字符
*****************************************************************
* rfind   从右向左找到第一个目标子串
* size_type rfind( CharT ch, size_type pos = 0 ) const;
* size_type rfind( const CharT* s, size_type pos = 0 ) const;
* size_type rfind( const CharT* s, size_type pos, size_type count) const;
* size_type rfind( const string& str, size_type pos = 0 ) const;
* 参数 pos:搜索起始下标位置，右到左，参数 count:子串首多少个字符
*****************************************************************
* find_first_of  string 对象中第一个属于指定字符串中任何字符的元素
* size_type find_first_of( CharT ch, size_type pos = 0 ) const;
* size_type find_first_of( const CharT* s, size_type pos = 0 ) const;
* size_type find_first_of( const CharT* s,
                           size_type pos, size_type count) const;
* size_type find_first_of( const string& str, size_type pos = 0 ) const;
*****************************************************************
* find_first_not_of  对象中第一个不属于指定字符串中任何字符的元素
* size_type find_first_not_of( CharT ch, size_type pos = 0 ) const;
* size_type find_first_not_of( const CharT* s, size_type pos = 0 ) const;
* size_type find_first_not_of( const CharT* s,
                               size_type pos, size_type count) const;
* size_type find_first_not_of( const string& str, size_type pos = 0 )
```

```
const;
*************************************************************************
* find_last_of  string 对象中最后一个属于指定字符串中任何字符的元素
* size_type find_last_of( CharT ch, size_type pos = 0 ) const;
* size_type find_last_of( const CharT* s, size_type pos = 0 ) const;
* size_type find_last_of( const CharT* s, size_type pos, size_type coun;
* size_type find_last_of( const string& str, size_type pos = 0 ) const;
*************************************************************************
* find_last_not_of  对象中最后一个不属于指定字符串中任何字符的元素
* size_type find_last_not_of( CharT ch, size_type pos = 0 ) const;
* size_type find_last_not_of( const CharT* s, size_type pos = 0 ) const;
* size_type find_last_not_of( const CharT* s,
                              size_type pos, size_type count) const;
* size_type find_last_not_of( const string& str, size_type pos = 0 )
*************************************************************************
* substr  返回一个子串
* string substr( size_type pos = 0, size_type count = npos ) const;
*************************************************************************
* starts_with  验证起始字符或字符串是否为参数指定的
* bool starts_with( CharT ch ) const;
* bool starts_with( const CharT* s ) const;
* ends_with    验证结尾字符或字符串是否为参数指定的
* bool ends_with( CharT ch ) const;
* bool ends_with( const CharT* s ) const;
* contains     验证结是否包含参数指定的字符或字符串
* bool contains( CharT ch ) const;
* bool contains( const CharT* s ) const;
*************************************************************************/
```