一、运算符重载

1. 什么是运算符重载

> 函数可以重载， 运算符也是可以重载的。 运算符重载是对已有的运算符重新进行定义，赋予其另一种功能，**以达到适应不同的数据类型**。运算符重载不能改变它本来的寓意(也就是加法不能变更为 减法)
>
> 运算符重载只是一种 “语法上的方便” ， 背后实际上是一种函数调用的方式。

```
#include<iostream>
using namespace std;
class Student{
    int age;
    public:
        Student(int age):age(age){
        }
};
int main(){
    int a = 3 ;
    int b = 4 ;

    //两个整数相加，这是允许的。
    int c = a + b ;

    Student s1(10) ;
    Student s2(20) ;

    //两个学生相加，则编译失败。即使你认为两个学生的年纪之和为第三个学生的年纪
    //依然不允许通过编译，因为 + 运算符在定义之初就不认识Student这种类型。
    Student s3 = s1 + s2 ; // 编译器不通过

    return 0 ;
}
```

2. 运算符重载的意义

> 运算符的重载实际上背后是调用对应的函数，重载运算符使得把复杂的代码包装起来，对外暴露简单的一个符号即可。实际上不使用运算符重载，也一样可以实现功能，如下面两个学生相加的示例所示：

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

```
3
4  class Student{
5
6  public:
7      int age;
8
9      Student(int age):age(age){
10
11     }
12
13 };
14
15 int main(){
16
17     Student s1(10);
18     Student s2(20);
19
20     //先对两个学生的年龄做加法
21     int age = s1.age + s2.age;
22
23     //再赋值给第三名学生
24     Student s3 (age);
25
26     cout << "第三名学生的年龄是：  " << s3.age << endl;
27
28     return 0 ;
29 }
```

3. 定义运算符

> 重载的运算符是带有特殊名称的函数，函数名是由关键字 `operator` 和其后要重载的运算符符号构成的。比如，要重载 `+` 运算符，那么可以声明一个函数为 `operator+()` ，函数声明的位置可以是类的内部，也可以是类的外部，所以又有了成员函数和全局函数的划分。与其他函数一样，重载运算符函数，也可以拥有返回值和参数列表。此处仍然以学生相加的案例举例。

1. 成员函数方式

> 把重载的运算符函数定义在类中，此时**只需要接收一个参数**，因为类的对象本身作为 `+` 的前面调用者。

```
1  #include<iostream>
2
3
4  using namespace std;
5
6  class Student{
7
```

```
public:
    int age;

    Student(int age):age(age){

    }

    s1 + s2;

    //两个学生的年龄之和，则为第三个学生的命令，所以此处需要返回一个学生对象。
    //好方便在外面接收。
    Student operator+ (Student &s ){
        Student temp(this->age + s.age);
        return temp;
    }

};

int main(){

    Student s1(10);
    Student s2(20);

    //这里等于使用s1的对象，调用了operator+这个函数， +后面的 s2 则被当成参数来传递
    Student s3 = s1 + s2;

    cout << "s3.age = " << s3.age << endl;

    return 0 ;
}
```

2. 全局函数方式

> 并不是所有的运算符重载都能定义在类中，比如，需要扩展一些已有的类，对它进行运算符重载，而这些类已经被打成了一个库来使用，此时通过全局函数的方式来实现运算符重载。

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

class Student{

public:
    int age;

    Student(int age):age(age){

    }
};

//由于函数并非定义在类中，所以此处无法使用到this指针，则需要传递两个对象进来。
```

```
Student operator+ (Student &s , Student &ss ){
Student temp(s.age + ss.age);
return temp;
}

int main() {

    Student s1(20);
    Student s2(30);

    //这里等于使用s1的对象，调用了operator+这个函数， +后面的 s2 则被当成参数来传递
    Student s3 = s1 + s2;


    cout << "s3.age = " << s3.age << endl;
    return 0;
}
```

练习

学生常常有自己的零用钱，一天张三和李四碰面，两人在合计计算他们的零用钱之和，请使用运算符重载的方式，设计他们的行为。

class stu{

int momey ;

4. 输出运算符重载

输出运算符重载，实际上就是 `<<` 的重载。 `<<` 实际上是位移运算符，但是在c++里面，可以使用它来配合 `cout` 进行做控制台打印输出。 `cout` 其实是 `ostream` 的一个实例，而 `ostrem` 是 类 `basic_ostream` 的一个别名，所以之所以能使用 `cout <<` 来输出内容，全是因为 `basic_ostream` 里面对 `<<` 运算符进行了重载.

```
<<` 运算符只能输出常见的基本数据类型，对于自定义的类型，是无法识别的。比如：`student
#include <iostream>
using namespace std;

class Student{
public:
   string name {"zhangsan"};

public:
   Student(string name){
       this->name = name;
   }
```

```
};

//对 << 运算符进行重载。 这里使用的是全局函数 ，第一个参数ostream 是因为在外面调用<<这个运算符的时候，cout 在前，cout 为ostream这种类型的对象。
ostream& operator<< (ostream& o, Student &s1){
    o << s1.name ;
    return o;
}


int main() {
    Student s1("张三");
    cout << s1  <<endl ;
    return 0;
}
```

5. 输入运算符重载

> 输入运算符重载，实际上就是 `>>` 的重载。用意和上面的输出运算符相似

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

class Student{
public:
   string name



//对 << 运算符进行重载。
ostream& operator<< (ostream& o, Student &s1){
    o << s1.name ;
    return o;
}

//对 >> 运算符进行重载
istream& operator>> (istream& in, Student &s1){
    in >> s1.name;
    return in;
}


int main() {

    Student s1;

    cout << "请输入学生的姓名:" << endl;
    cin >>  s1;

    //打印学生， 实际上是打印他的名字。
```

```
31      cout << s1  <<endl ;
32      return 0;
33  }
```

练习

> 学生类中有姓名和年龄属性，请重载输入运算符，以便从键盘直接获取姓名和年龄。形如：
>
> stu s ;
>
> cin >> s; //键盘录入 姓名 年龄 （ 中间有空格区分)

6.赋值运算符重载【了解】

1. 默认的赋值运算符

> 赋值运算符在此前编码就见过了，其实就是 `=` 操作符 。

```
1   class Student{
2       int no;
3       int age ;
4   
5   public :
6       Student(no , age){
7           this->no = no ;
8           this->age = age ;
9       }
10  }
11  
12  int main(){
13  
14      Student s1(10001 , 15);
15      Student s2 ;
16      s2 = s1; //此处使用了默认的赋值运算符。
17  
18  
19      Student s2 = s1;//此处执行的是拷贝构造函数
20  
21      return 0 ;
22  }
```

2. 拷贝赋值运算

> 其实这里说的就是 `=` 赋值运算符

```
#include<iostream>

using namespace std;

class Student{
    int no;
    int age ;

public :
    Student(no , age){
        this->no = no ;
        this->age = age ;
    }

    //拷贝赋值
    Stu& operator=(const Stu &h){
        cout <<"执行拷贝赋值函数" << endl;
        d = new int();
        *d = *h.d;
    }
}


int main(){
    Stu stu1("张三",18);
    Stu stu2 ;

    stu2 = stu1; //拷贝赋值

    return 0 ;
}
```

7. 调用运算符重载【了解】

> 一般来说，可以使用对象来访问类中的成员函数，而对象本身是不能像函数一样被调用的，除非在类中重载了`调用运算符`。 如果类重载了函数调用运算符，则我们可以像使用函数一样使用该类的对象。在外面使用 `对象()`,实际上背后访问的是类中重载的调用运算符函数。
>
> 如果某个类重载了调用运算符，那么该类的对象即可称之为： ***函数对象*** ，因为可以调用这种对象，所以才说这些对象行为 `像函数一样`。

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

class Calc{
```

```
public:
    int operator()(int val){
        return val <0 ? -val :val;
    }
};
int main(){
    Calc c ;
    int value = c(-10);
    return 0;
}

```

- 标准库中的函数对象

> 在标准库中定义了一组算术运算符 、关系运算符、逻辑运算符的类，每个类都有自己重载的调用运算符。要想使用这些类，需要导入 `#include` ， 后面要说的 `lamdda表达式` 正是一个函数对象

```
#include<functional>

int main(){

    plus<int > p; //加法操作
    int a = p(3 , 5);

    negate<int> n; //可以区绝对值
    cout <<n(-10) << endl;

    return 0 ;
}
```

二、lambda表达式

1. lambda入门

> 也叫做 `lambda 函数` , lambda 表达式的出现目的是为了提高编码效率，但是它的语法却显得有点复杂 ,并且阅读性比较差。lambda表达式表示一个可以执行的代码单元，可以理解为一个未命名的内联函数。

1. 表达式的语法

> **在编写lambda表达式的时候，可以忽略参数列表和返回值类型，但是前后的捕获列表和函数体必须包含** ， 捕获列表的中括号不能省略，编译根据它来识别后面是否是 `lambda表达式` ，并且它还有一个作用是能够让lambda的函数体访问它所处作用域的成员。

```
//语法
[捕获列表](参数列表)->返回值类型{函数体}
```

2. 简单示例

- 示例1

```
//示例1:
[](int a ,int b)->int{return a + b ;} ; //一个简单的加法
```

- 示例2

> 如果编译器能推断出来类型。，那么 ->int 也可以省略掉

```
[](int a ,int b){return a + b ;} ;
```

- 示例3

> 如果不需要参数，那么参数列表页可以忽略。至此不能再精简了。

```
[]{return 3 + 5 ;};
```

- 示例4

> 这是最精简的lambda表达式了，不过没有任何用处，等于一个空函数，没有函数体代码

```
[]{};
```

2. lambda的使用

> `lambda表达式` 定义出来并不会自己调用，需要手动调用。 如下面所示的，一个加法的案例

- 使用变量接收，然后再调用

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

int main(){
    //1. 接收lambda表达式，然后调用
    auto f = [](int a ,int b)->int{return a + b ;};

    //2. 调用lambda表达式
    int result = f(3,4); //调用lambda函数，传递参数

    cout << "result = " << result << endl;
    return 0 ;
}
```

- 不接受表达式，直接调用

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

int main(){
    ///2. 不接收，立即调用。 后面的小括号等同于调用这个函数。
    int result= [](int a ,int b){return a + b }(3,4);

    cout << "result = " << result << endl;
    return 0 ;
}
```

3. 捕获列表的使用

> labmda表达式需要在函数体中定义，这时如果想访问所处函数中的某个成员，那么就需要使用捕获列表了。捕获列表的写法通常有以下几种形式：

形式	作用
[a]	表示值传递方式捕获变量 a
[=]	表示值传递方式捕获所有父作用域的变量（包括this）
[&a]	表示引用方式传递捕获变量a
[&]	表示引用传递方式捕获所有父作用域的变量（包括this）
[this]	表示值传递方式捕获当前的this指针
[=,&a,&b]	引用方式捕获 a 和 b , 值传递方式捕获其他所有变量 （这是组合写法）

```cpp
#include<iostream>

using namespace std;

int main(){

    int a = 3 ;
    int b = 5;


    //值传递方式捕获 a 和 b ， 在lanmbda 表示内部无法修改a和b的值
    auto f1 = [a,b]{return a + b;};
    cout << f1()  << endl; //打印 8


    //引用方式捕获 a 和 b  ，可以在内部修改a 和 b的值
    auto f2 = [&a,&b]{
        a = 30; //这里修改会导致外部的a 也跟着修改。
        return a + b;
    };

```

```
    cout << f2()  << endl; //这里打印35
    cout << "a= "<< a << endl; //再打印一次，a 变成30了

    return 0 ;
}
```

4. lambda 的应用场景

> 编写lamdda表达式很简单，但是用得上lambda表达式的地方比较特殊。一般会使用它来封装一些逻辑代码，使其不仅具有函数的包装性，也具有可见的自说明性。

> 在C++ 中，函数的内部不允许在定义函数，如果函数中需要使用到某一个函数帮助计算并返回结果，代码又不是很多，那么lambda表达式不失为一种上佳选择。如果没有lambda表达式，那么必须在外部定义一个内联函数。 来回查看代码稍显拖沓，定义lambda函数，距离近些，编码效率高些。

> lambda表达式其实就是一个内联函数。

1. 没有使用lambda函数

> 计算6科考试总成绩。

```
#include<iostream>

using namespace std;

int getCout(vector<int> scores){
    int result = 0 ;
    for(int s : scores){
        result += s;
    }
    return result;
}

int main(){
    vector<int> scores{80,90,75,99,73,23};
    //获取总成绩
    int result = getCout(scores);
    cout <<"总成绩是:  "<< result << endl;

    return 0 ;
}
```

2. 使用labmda表达式

> lambda函数属于内联，并且靠的更近，也便于阅读。

```
#include<iostream>

using namespace std;

int main(){

    vector<int> scores{80,90,75,99,73,23};
    int result2 = [&]{
        int result = 0 ;
        for(int s : scores){
            result += s;
        }
        return result;
    }();
    cout <<"总成绩是2：  "<< result2 << endl;
    return 0 ;
}
```

练习：

> 现在要比较两个学生的大小，可以按照名字多少排序比较、也可以按照年龄比较、也可以按照身高比较。定义一个全局函数compare，接收三个参数，参数一和参数二 为要比较的学生对象，参数三为比较逻辑的函数（此应该使用lambda表达式来写，根据lambda表达式的返回值来决定谁大谁小。）

三、 继承

1. 什么是继承

> 继承是类与类之间的关系，是一个很简单很直观的概念，与现实世界中的继承类似，例如儿子继承父亲的财产。
>
> 继承（Inheritance）可以理解为一个类从另一个类获取成员变量和成员函数的过程。例如B类 继承于A类，那么 B 就拥有 A 的成员变量和成员函数。被继承的类称为父类或基类，继承的类称为子类或派生类。 子类除了拥有父类的功能之外，还可以定义自己的新成员，以达到扩展的目的。

1. is A 和 has A

> 大千世界，不是什么东西都能产生继承关系。只有存在某种联系，才能表示有继承关系。如：哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。所以在学习继承后面的内容之前，先说说两个术语 `is A` 和 `has A`。

- is A

> 是一种继承关系，指的是类的父子继承关系。表达的是一种方式：这个东西是那个东西的一种。例如：长方体与正方体之间--正方体是长方体的一种。正方体继承了长方体的属性，长方体是父类，正方体是子类。

- has A

> has-a 是一种组合关系，是关联关系的一种（一个类中有另一个类型的实例），是整体和部分之间的关系（比如汽车和轮胎之间），并且代表的整体对象负责构建和销毁部分对象，代表部分的对象不能共享。

2. 继承入门

> 通常在继承体系下，会把共性的成员，放到父类来定义，子类只需要定义自己特有的东西即可。或者父类提供的行为如不能满足，那么子类可以选择自定重新定义。

```
#include <iostream>
#include <string>

using namespace std;

//父类
class Person{
    public:
        string name;
        int age ;
};


//子类
class Student:public Person{

};

int main(){

    //子类虽然没有声明name 和 age ，但是继承了person类，等同于自己定义的效果一样
    Student s;
    s.name = "张三";
    s.age = 18;

    cout << s.name << " = " << s.age << endl;
```

```
27
28
29      return 0 ;
30  }
```

3. 访问权限回顾

> 当一个类派生自基类，该基类可以被继承为 **public、protected** 或 **private** 几种类型。继承类型是在继承父类时指定的。 如： `class Student : public Person` 。 我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。

- public

> 表示公有成员，该成员不仅可以在类内可以被访问，在类外也是可以被访问的，是类对外提供的可访问接口；

- private

> 表示私有成员，该成员仅在类内可以被访问，在类的外面无法访问；

- protected

> 表示保护成员，保护成员在类的外面同样是隐藏状态，无法访问。但是可以在子类中访问。

1. 公有继承（public）

> 基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限；

```
1   #include <string>
2   using namespace std;
3
4   class person{
5       public:
6           string name;
7
8       private:
9           int age;
10  };
11
12  class student:public person{
13      //name 和 age保持原有访问权限。
14  };
15
16  int main(){
17
18      student s;
19      s.name = "张三" ;
```

```
    s.age = 18 ; //编译错误！ 无法访问 age

    return 0 ;
}
```

2. 私有继承（private）

> 基类所有成员在派生类中的访问权限都会变为私有(private)权限；

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

class person{
    public:
        string name;

    private:
        int age;
};

class student:private person{
    //name 和 age保持全部变成private权限
};

int main(){

    student s;
    s.name = "张三" ;//编译错误！ 无法访问 name
    s.age = 18 ; //编译错误！ 无法访问 age

    return 0 ;
}
```

3. 保护继承（protected）

> 基类的共有成员和保护成员在派生类中的访问权限都会变为保护(protected)权限，在子类中具有访问权限，但是在类的外面则无法访问。

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

class person{
public:
    string name;

private:
    int age;
};

class student: protected person{
```

```
    //name 和 age保持原有访问权限。

public:
    void read(){
        s.name = "李四";
        s.age = 19 ;
        cout << s.age << " 的 " << s.name << " 在看书";
    }
};

int main(){

    student s;
    s.read();

    //类的外面无法访问，编译错误。
    s.name = "张三" ;
    s.age = 18 ;

    return 0 ;
}
```

4. 构造和析构

1. 继承状态

> 构造函数是对象在创建是调用，析构函数是对象在销毁时调用。但是在继承关系下，无论在对象的创建还是销毁，都会执行父类和子类的构造和析构函数。 它们一般会有以下规则：

> a. 子类对象在创建时会首先调用父类的构造函数;
>
> b. 父类构造函数执行完毕后,执行子类的构造函数;
>
> c. 当父类的构造函数中有参数时,必须在子类的初始化列表中显示调用;
>
> d. 析构函数执行的顺序是先调用子类的析构函数，再调用父类的析构函数

```
#include <iostream>

using namespace std;

class person{
public :
    person(){
        cout << "调用了父类构造函数" << endl;
    }
    ~person(){
        cout << "调用了父类析构函数" << endl;
    }
}
```

```
14
15  class student: public person{
16  public :
17      student(){
18          cout << "调用了子类类构造函数" << endl;
19      }
20      ~student(){
21          cout << "调用了子类析构函数" << endl;
22      }
23  };
24
25
26  int main() {
27      Student s1
28      return 0;
29  }
```

2. 继承和组合

> 如果在继承状态下，子类中的成员又含有其他类的对象属性，那么他们之间的构造很析构调用顺序，遵循以下原则：
>
> a. 先调用父类的构造函数,再调用组合对象的构造函数,最后调用自己的构造函数;
>
> b. 先调用自己的析构函数,再调用组合对象的析构函数,最后调用父类的析构函数。

```
1   #include <iostream>
2
3   using namespace std;
4
5   //父类
6   class Person{
7
8   public :
9
10      Person(){
11          cout << "调用了父类构造函数" << endl;
12      }
13
14      ~Person(){
15          cout << "调用了父类析构函数" << endl;
16      }
17
18  };
19
20  //其他类
21  class  A{
22  public :
23      A(){
24         cout << "调用A的构造函数" << endl;
25      }
26
```

```
27      ~A(){
28          cout << "调用A的析构函数" << endl;
29      }
30  };
31
32  //子类
33  class Student: public Person{
34
35  public :
36      Student(){
37          cout << "调用了子类类构造函数" << endl;
38      }
39
40      ~Student(){
41          cout << "调用了子类析构函数" << endl;
42      }
43  public:
44      A a;
45  };
46
47
48  int main() {
49      Student s1(18 , "zhangsan");
50      return 0;
51  }
```

3. 调用父类有参构造

> 继承关系下，子类的默认构造函数会隐式调用父类的默认构造函数，假设父类没有默认的无参构造函数，那么子类需要使用参数初始化列表方式手动调用父类有参构造函数。
>
> 一般来说在创建子类对象前，就必须完成父类对象的创建工作，也就是在执行子类构造函数之前，必须先执行父类的构造函数。c++ 使用初始化列表来完成这个工作

- 父类

```
1   #include <iostream>
2
3   using namespace std;
4
5   class Person{
6
7   private :
8       int age ;
9       string name ;
10
11  public :
12      Person(int age , string name){
13          cout << "调用了父类构造函数" << endl;
14          this->age = age ;
15          this->name = name;
16      }
```

```
};
```

- 子类

> 子类只能使用初始化列表的方式来访问父类构造

```
class Student: public Person{

public :
    Student(int age , string name):Person(age ,name){
        cout << "调用了子类类构造函数" << endl;
    }
};

int main(){
    Student s1(18 , "zs");
    return 0 ;
}
```

4. 再说初始化列表

> 初始化列表在三种情况下必须使用:
>
> - 情况一、需要初始化的数据成员是对象，并且对应的类没有无参构造函数
> - 情况二、需要初始化const修饰的类成员或初始化引用成员数据;
> - 情况三、继承关系下，父类没有无参构造函数情况
>
> 初始化列表的赋值顺序是按照类中定义成员的顺序来决定

1. 常量和引用的情况

> 类中成员为 `引用` 或者 `const修饰` 的成员

```
#include <iostream>
#include <string>

using namespace std;

class stu{

public:
    const string name; //常量不允许修改值，所以不允许在构造里面使用  = 赋值
    int &age; //

    stu(string name , int age):name(name),age(age){
        cout << "执行构造函数" <<endl;
    }
};

int main(){

```

```
    stu s1("张三" , 88);
    cout << s1.name << " = " << s1.age << endl;

    return 0 ;
}
```

2. 初始化对象成员

> 类中含有其他类的对象成员，如果要初始化，只能使用初始化类列表方式。

```
#include <iostream>

using namespace std;

class A{

public:
    int number;
    A(int number):number(number){
        cout << "执行了A的构造函数" <<endl;
    }
};

class stu{
public:
    A a;

    stu():a(9){

        cout << "执行了stu的构造函数" <<endl;
    }
};

int main(){
    stu s;
    return 0;
}
```

5. 重写父类同名函数

> 在继承中，有时候父类的函数功能并不够强大，子类在继承之后，可以对其进行增强扩展。
> 如果还想调用你父类的函数，可以使用 `父类::函数名()` 访问

```
#include <iostream>

using namespace std;

class WashMachine{
public:
```

```
    void wash(){
        cout << "洗衣机在洗衣服" << endl;
    }
};


class SmartWashMachine : public WashMachine{
public:
    void wash(){
        cout << "智能洗衣机在洗衣服" << endl;
        cout << "开始添加洗衣液~~" << endl;
        //调用父类的函数
        WashMachine::wash();
    }
};

int main(){

    SmartWashMachine s;
    s.wash();

    return 0 ;
}
```

6. 多重继承

> C++ 允许存在多继承，也就是一个子类可以同时拥有多个父类。只需要在继承时，使用逗号进行分割即可。

```
using namespace std;

class Father{
public:
    void makeMoeny(){
        cout << "赚钱" << endl;
    }
};
class Mother{
public:
    void makeHomeWork(){
        cout << "做家务活" << endl;
    }
};

class Son:public Father , public Mother{

};

int main(){

```

```
    Son s ;
    s.makeMoeny();
    s.makeHomeWork();

    return 0 ;
}
```

1. 多重继承的构造函数

> 多继承形式下的构造函数和单继承形式基本相同，只是要在子类的构造函数中调用多个父类的构造函数 。 他们调用的顺序由定义子类时，继承的顺序决定。

```
#include <iostream>

using namespace std;

class Father{

    string name;
public:
    Father(string name):name(name){
        cout << "执行父亲构造函数" <<endl;
    }
};



class Mother{
    int age;

public:
    Mother(int age):age(age){
        cout << "执行母亲构造函数" <<endl;
    }

};
class Son:public Father , public Mother{

public:
    Son(string name ,int age):Father(name),Mother(age){
        cout << "执行孩子构造函数" <<endl;
    }
};

int main(){

    Son s("无名氏" ,38);

    return 0 ;
}
```

7. 类的前置声明

> 一般来说，类和 变量是一样的，必须先声明然后再使用，如果在某个类里面定义类另一个类的对象变量，那么必须在前面做前置声明，才能编译通过。

```
class father; //所有前置声明的类，在某个类中定义的时候，只能定义成引用或者指针。

class son{
public:
    //father f0; //因为这行代码，单独拿出来说，会执行B类的无参构造，
    //但是编译器到此处的时候，还不知道B这个类的构造长什么样。
    father &f1;
    father *f2;

    son(father &f1 , father *f2):f1(f1),f2(f2){

    }
};


class father{

};


int main(){

    //  father b; //---> 执行B的构造函数。
    father f1;
    father f2;

    son s(f1 ,&f2);

    return 0 ;
}
```