我i们主要是在学用C++设计程序，其中最主要的编程范式就是面向对象。在面向对象的编程范式中，最主要的是学习三个模块：1.封装 ：属性跟方法封装到自定义的类型中，需要注意访问控制权限；区分类方法和成员方法；以及重载中的函数重载和运算符重载，在运算符重载中又引伸出三个重要对象：指针对象、数组对象、函数对象。有了这些概念以后，我们又深化地去应用了STL中源码方面地技巧。

封装：我该有的（属性）和我该做的（方法）。
继承：叫一声爸爸，开启财富之门
多态：我就是我，是不一样的烟火

继承

程序实现中涉及到**概念上的递进关系**的：猫和狗在动物层面应该拥有相同的特质（都有名字.....）。所以与其单独封装，倒不如把其中共同的部分独立出来然后共同继承自它。好处：1.减少编码量 2.逻辑清晰

```
class Animal {
public:
    string name() { return this->__name; }
private:
    string __name;
};
class Cat : public Animal {
};
```

基类：基础类型。（Animal）
派生类：由基类派生出来的类型。（Cat）

父类中的private：子类对于父类来说属于类外；private只能在类内或者友元函数中访问。所以子类不能访问父类中的private，只能访问`public`和`protected`权限。

外界想要访问子类中继承自父类的属性跟方法的时候：需要通过两道检票口：父类权限和继承权限。两者当中取更严格那个权限作为最终对外的权限。

子类会完完整整地继承父类地所有东西：虽然父类中私有的部分子类访问不到，但是那块内存依旧会存在子类的对象里。就像看不到不代表不存在一样。

子类继承自父类，**对应父类这片区域的初始化，应当交给父类的构造函数**，语法类似如下所示：

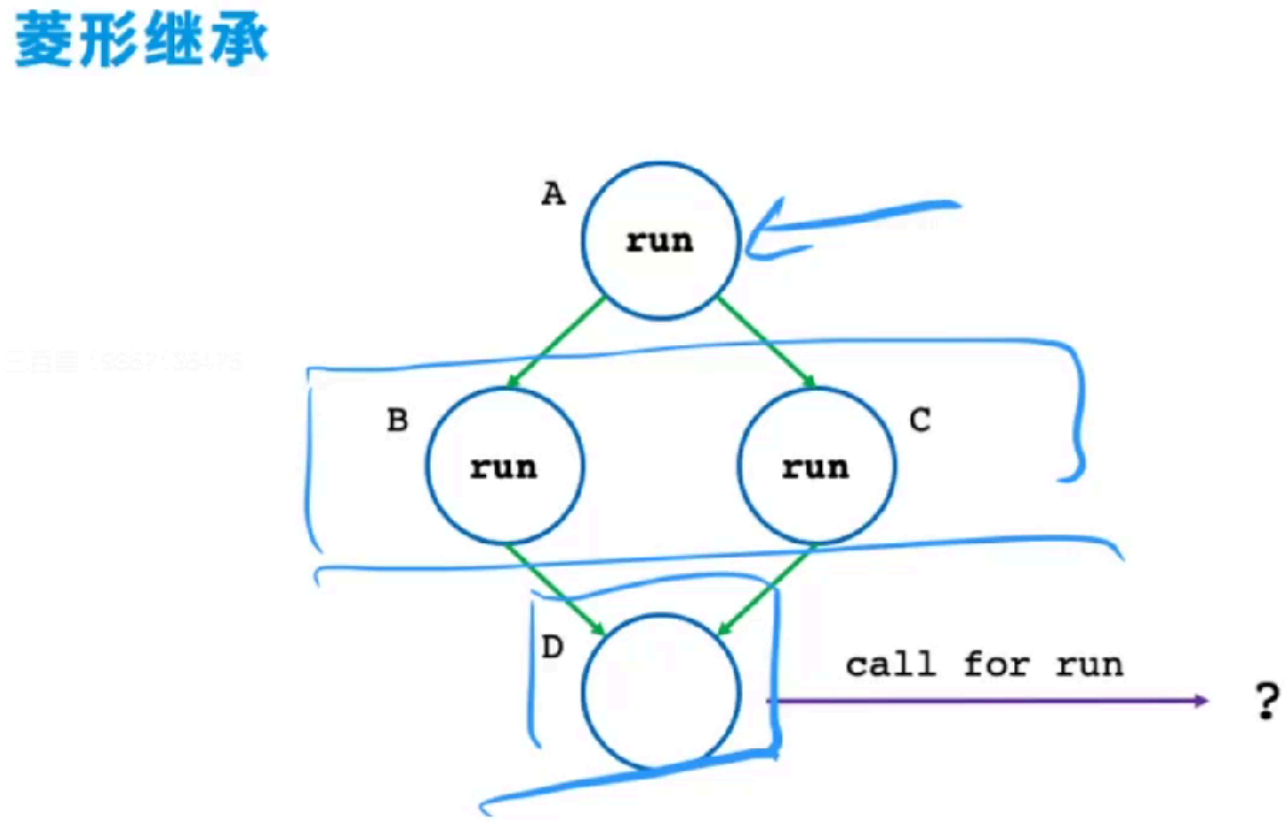
```
class A : public Base {
public:
        A() : Base("class_A") {}//可以在初始化列表中显性地调用父类构造函数，相当于指定用什么来构造父类那
};
```

子类到父类之间存在**隐式类型转换**：从概念上讲，子类也还是父类（人也是一种动物....）。但是父类到子类，从逻辑上讲是说不通的，所以不存在隐式类型转换。

```
class Base {
public:
    Base(string name) : class_name(name) {}
    int x;
    string class_name;
private:
    int y;
};
class A : public Base {
public:
    A() : Base("class_A") {}
};
void func(Base &b) { //引用
    cout << "input class : " << b.class_name << endl;
}
void func1(Base *b) { //指针
    cout << "input class : " << b.class_name << endl;
}
int main() {
    A a;
    func1(&a);
    func(a); // A类继承自Base类，所以a对象可以传给Base的引用或者指针—隐式类型转换
}
```

构造顺序：先完成基类的构造（调用父类的构造函数），在完成子类的构造。

菱形继承



编译器也许能编译通过，但不保证结果。也就是说不确定D中的run方法到底是哪里的run方法。此种情况应当尽量避免。

拷贝&赋值——继承

在设计**拷贝行为**的时候，最起码能马上反应过来两个部分：**拷贝构造函数**——构造阶段完成的拷贝行为；**赋值运算符**——出了构造阶段以后所调用的等号，它也是一类拷贝行为。

一定是**先完成父类的拷贝行为，再完成子类的拷贝行为**。

凡是子类中我们自己实现的拷贝行为，所有属性的拷贝需要显性地实现。：否则默认它会调用父类的默认构造。而不是父类的拷贝构造。

```
13  };
14  class ATTR1 : public ATTR_BASE {
15  public:
16      ATTR1(string name = "none") : ATTR_BASE(name) {}
17  };
18  class ATTR2 : public ATTR_BASE {
19  public:
20      ATTR2(string name = "none") : ATTR_BASE(name) {}
21  };
22  class Base {
23  public:
24      Base() : attr1("attr1 in Base"), attr2("attr2 in Base") {
25          cout << "Base constructor done" << endl;
26      }
27      Base(const Base &b) : attr1(b.attr1), attr2(b.attr2) {
28          cout << "Base copy constrcutor done" << endl;
29      }
30
31  private:
32      ATTR1 attr1;
33      ATTR2 attr2;
34  };
35
36  class A : public Base {
37  public:
38      A() {
39          cout << "A constructor done" << endl;
40      }
41      A(const A &a) {
```

父类的默认构造

父类的拷贝构造

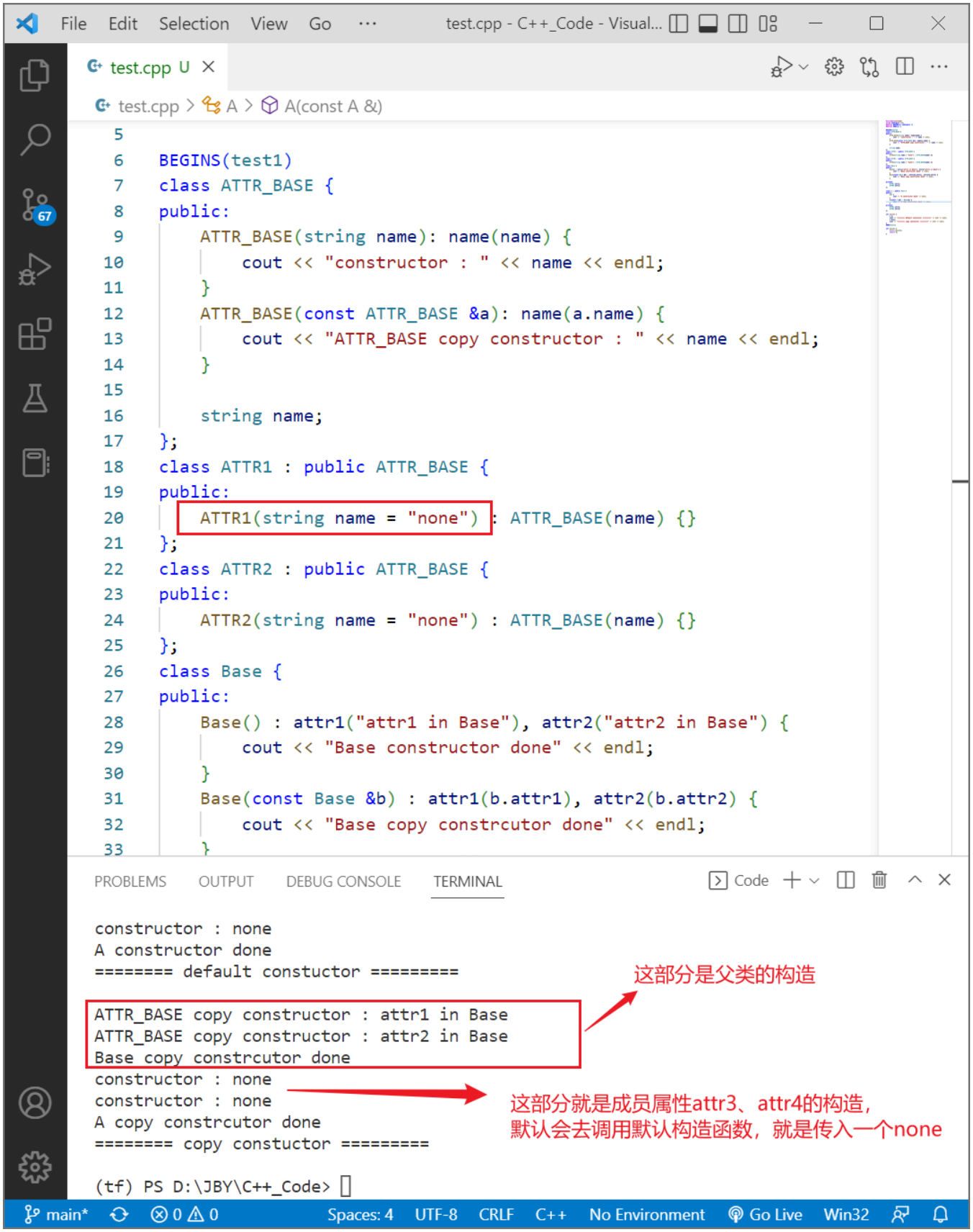
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Code

```
constructor : none
constructor : none
A constructor done
======== default constuctor =========

constructor : attr1 in Base
constructor : attr2 in Base
Base constructor done
constructor : none
constructor : none
======== copy constuctor =========
```

一旦没有显性地指明，输出的会是默认构造，而非拷贝构造。

这也是为什么我们需要在我们子类的拷贝构造函数中显示地指出用父类的拷贝构造去构造父类那片区域，避免错误


File Edit Selection View Go ···
test.cpp - C++_Code - Visual...
test.cpp U
test.cpp > A > A(const A &)
BEGINS(test1)
class ATTR_BASE {
public:
ATTR_BASE(string name): name(name) {
cout << "constructor : " << name << endl;
}
ATTR_BASE(const ATTR_BASE &a): name(a.name) {
cout << "ATTR_BASE copy constructor : " << name << endl;
}
string name;
};
class ATTR1 : public ATTR_BASE {
public:
ATTR1(string name = "none") : ATTR_BASE(name) {}
};
class ATTR2 : public ATTR_BASE {
public:
ATTR2(string name = "none") : ATTR_BASE(name) {}
};
class Base {
public:
Base() : attr1("attr1 in Base"), attr2("attr2 in Base") {
cout << "Base constructor done" << endl;
}
Base(const Base &b) : attr1(b.attr1), attr2(b.attr2) {
cout << "Base copy constrcutor done" << endl;
}
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
Code
constructor : none
A constructor done
======== default constuctor =========
ATTR_BASE copy constructor : attr1 in Base
ATTR_BASE copy constructor : attr2 in Base
Base copy constrcutor done
constructor : none
constructor : none
A copy constrcutor done
======== copy constuctor =========
(tf) PS D:\JBY\C++_Code>
这部分是父类的构造
这部分就是成员属性attr3、attr4的构造，
默认会去调用默认构造函数，就是传入一个none
main* 0 0 Spaces: 4 UTF-8 CRLF C++ No Environment Go Live Win32

```
#include<iostream>
using namespace std;
#define BEGINS(x) namespace x{
#define ENDS(x) }

BEGINS(test1)
class ATTR_BASE {
public:
    ATTR_BASE(string name): name(name) {
        cout << "constructor : " << name << endl;
    }
    ATTR_BASE(const ATTR_BASE &a): name(a.name) {
        cout << "ATTR_BASE copy constructor : " << name << endl;
    }
    ATTR_BASE &operator=(const ATTR_BASE &a) {
        name = a.name;
        cout << "operator= : " << name << endl;
        return *this;
    }
    ~ATTR_BASE() {
        cout << "destructor : " << name << endl;
    }
    string name;
};
class ATTR1 : public ATTR_BASE {
public:
    ATTR1(string name = "none") : ATTR_BASE(name) {}
};
class ATTR2 : public ATTR_BASE {
public:
    ATTR2(string name = "none") : ATTR_BASE(name) {}
};
class Base {
public:
    Base() : attr1("attr1 in Base"), attr2("attr2 in Base") {
        cout << "Base constructor done" << endl;
    }
    Base(const Base &b) : attr1(b.attr1), attr2(b.attr2) {
        cout << "Base copy constrcutor done" << endl;
    }
    Base &operator=(const Base &b) {
        attr1 = b.attr1;
        attr2 = b.attr2;
        cout << "Base operator= done" << endl;
        return *this;
    }
```

```
    ~Base() {
        cout << "Base destructor done" << endl;
    }
private:
    ATTR1 attr1;
    ATTR2 attr2;
};

class A : public Base {
public:
    A() : Base(), attr3("attr3 in A"), attr4("attr4 in A") {
        cout << "A constructor done" << endl;
    }
    A(const A &a) : Base(a), attr3(a.attr3), attr4(a.attr4) {
        cout << "A copy constrcutor done" << endl;
    }
    A &operator=(const A &a) {
        this->Base::operator=(a);
        attr3 = a.attr3;
        attr4 = a.attr4;
        cout << "A operator= done" << endl;
        return *this;
    }
    ~A() {
        cout << "A destructor done" << endl;
    }
private:
    ATTR1 attr3;
    ATTR2 attr4;
};

int main() {
    A a;
    cout << "======== default constuctor =========" << endl << endl;
    A b(a);
    cout << "======== copy constuctor =========" << endl << endl;
    b = a;
    cout << "======== operator assign =========" << endl << endl;
    cout << "======== destructor =========" << endl;
    return 0;
}
ENDS(test1)

int main() {
    test1::main();
    return 0;
```

```
}
```

```
47      ATTR2 attr2;
48  };
49
50  class A : public Base {
51  public:
52      A() {
53          cout << "A constructor done" << endl;
54      }
55      A(const A &a) : Base(a), attr3(a.attr3), attr4(a.attr4) {
56          cout << "A copy constrcutor done" << endl;
57      }
58      A &operator=(const A &a) {
59          this->Base::operator=(a);
60          // attr3 = a.attr3;
61          // attr4 = a.attr4;
62          cout << "A operator= done" << endl;
63          return *this;
64      }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL Code

```
Base copy constrcutor done
ATTR_BASE copy constructor : none
ATTR_BASE copy constructor : none
A copy constrcutor done
======== copy constuctor =========

operator= : attr1 in Base
operator= : attr2 in Base
Base operator= done
A operator= done
======== operator assign =========
```

这块我的父类的属性赋值完了，但是子类自己的属性还需要调用相应的赋值运算符，不然这部分内存是不会自己复制过来的。