卷纸

2021年6月4日 16:47

什么是二义性：

在多继承的场景里，当父类中存在**同名变量**时，子类访问父类的**同名变量**，将出现二义性，因为编译器不知道你将要访问的是哪个父类中的变量。

如何解决二义性：

1. 不使用多继承
2. 使用虚继承

```
class A
{
public:
    int a;
};
class B1 : virtual public A // 虚继承
{
};
class B2 : virtual public A // 虚继承
{
};
class C : public B1, public B2
{
};

int main()
{
    C c;
    c.a = 10; // OK，不会有二义性了
    return 0;
}
```

3. 使用类名::变量名

```
class B1
{
public:
    int a;
};
class B2
{
public:
    int a;
};
class C : public B1, public B2
{
};

int main()
{
    C c;
    c.B1::a = 10;
    c.B2::a = 20;
    return 0;
}
```

二义性的例子：

```
class A
{
public:
    int a; // B1，B2 都将继承一个变量 a
};
class B1 : public A
{
};
class B2 : public A
{
};
class C : public B1, public B2
{
};

int main()
{
    C c;
    c.a = 10; // ERROR ！ 二义性 ！！！
    return 0;
}
```

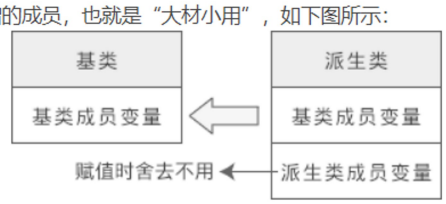
赋值兼容性规则：

赋值兼容规则中所指的替代包括以下情况：

(1)派生类对象可以赋值给基类对象

A中有一个成员是m_A，B继承A，B新增成员m_B
A a(10); B b(66,99);
此时A中m_A为10，B中m_A为66，m_B为99
如果a=b
那么A中m_A变为66

将派生类对象赋值给基类对象时，会舍弃派生类新增的成员，也就是“大材小用”，如下图所示：



可以发现，即使将派生类对象赋值给基类对象，基类对象也不会包含派生类的成员，所以依然不同通过基类对象来访问派生类的成员。对于上面的例子，a.m_a 是正确的，但 a.m_b 就是错误的，因为 a 不包含成员 m_b。

这种转换关系是不可逆的，只能用派生类对象给基类对象赋值，而不能用基类对象给派生类对象赋值。理由很简单，基类不包含派生类的成员变量，无法对派生类的成员变量赋值。同理，同一基类的不同派生类对象之间也不能赋值。

(2)派生类对象可以初始化基类的引用

修改上例中 main() 函数内部的代码，用引用取代指针：

```
int main(){
    D d(4, 40, 400, 4000);

    A &ra = d;
    B &rb = d;
    C &rc = d;

    ra.display();
    rb.display();
    rc.display();

    return 0;
}
```

运行结果：
Class A: m_a=4
Class B: m_a=4, m_b=40
Class C: m_c=400

(3)派生类对象的地址可以赋值给指向基类的指针

A是基类，有m_A
B继承A，新增m_B
C是另外一个基类，有m_C
D继承B和C，新增m_D

```
int main(){
    A *pa = new A(1);
    B *pb = new B(2, 20);
    C *pc = new C(3);
    D *pd = new D(4, 40, 400, 4000);

    pa = pd;
    pa -> display();

    pb = pd;
    pb -> display();

    pc = pd;
    pc -> display();

    cout<<"----------------------"<<endl;
    cout<<"pa="<<pa<<endl;
    cout<<"pb="<<pb<<endl;
```

运行结果：
Class A: m_a=4
Class B: m_a=4, m_b=40
Class C: m_c=400
----
pa=0x9b17f8
pb=0x9b17f8
pc=0x9b1800
pd=0x9b17f8

编译器通过指针来访问成员变量，指针指向哪个对象就使用哪个对象的数据；
编译器通过指针的类型来访问成员函数，指针属于哪个类的类型就使用哪个类的函数

```
    cout<<"----------------------"<<endl;
    cout<<"pa="<<pa<<endl;
    cout<<"pb="<<pb<<endl;
    cout<<"pc="<<pc<<endl;
    cout<<"pd="<<pd<<endl;

    return 0;
}
```

在替代之后，派生类对象可以作为基类对象使用，但是只能使用从基类继承来的成员。

四个区：

1. 代码区
   存放程序代码
2. 静态存储区/全局区
   存放静态变量、全局变量
3. 堆区
   存放new申请的空间
4. 栈
   存放局部变量、函数参数等

用new来创建对象的分类：

A *p1 = new A;　　//调用默认构造函数
A *p2 = new A(2);　//调用A (int i)
A *p3 = new A(“xyz”);　//调用A (char*)
A *p4 = new A[20];　　//创建动态数组时只能**调用各对象的默认构造函数**！！

const小结：

const int a = 10;　//用const定义的常量必须在定义时初始化
int * p;
p = &a;　　//const int *不能赋值给int *

const int * q = 8;　//q指向的值不可改变
int a = 8;
int * const m = &a;
*m = 9;　　//可以
int b = 9;
m = &b;　　//不行，因为m的指向不可以改变

const int * const p = a;　//p的指向和值都不可以改变

函数重载：

函数名必须一样，但是要有不同的参数，不同的参数是指参数的类型或个数有所不同
返回值不同不是函数重载

如果一个类D既有基类B、又有成员对象类M，则

- 在创建D类对象时，构造函数的执行次序为B->M->D
- 当D类的对象消亡时，析构函数的执行次序为D->M->B

多继承，比如D同时继承B、C
那么在调用时的顺序是看继承的顺序，而不是调用的顺序

对于拷贝构造函数：

- 派生类的隐式拷贝构造函数（由编译程序提供）将会调用基类的拷贝构造函数。
- 派生类自定义拷贝构造函数在默认情况下则调用基类的默认构造函数。需要时，可在派生类自定义拷贝构造函数的“基类成员初始化表”中显式地指出调用基类的拷贝构造函数。

拷贝构造函数的调用时机：

1. 定义对象：
   A a1;
   A a2(a1); 或 A a2 = a1; 或 A a2 = A(a1);
2. 把对象作为值传递给参数
3. 把对象作为函数的返回值

this指针的目的：
保证每个对象拥有自己的数据成员，但共享处理这些数据的代码。

```
template<class T>
T func(T x, T y)
{
    return x * x + y * y;
}
```

func(3, 5)	对
func(3.5, 5.0)	对
func<int>(3.5, 5)	对（显式实例化）
func<double>(3.5, 5.0)	对（同上）
func(3.5, 5)	错（没有对应的重载函数）

在C++中，输入/输出**不是语言定义的成分**，而是由具体的实现作为标准的功能来提供的
在C++中，输入/输入是一种**基于字节流**的操作
输入：把输入的数据看做是逐个字节的从外设流入计算机内存
输出：把输出的数据看做是逐个字节的从内部流出到外设

myswap的几种形式

```
void myswap(int &x, int &y)
{
    int t;
    t = x;
    x = y;
    y = t;
}

int main()
{
    int a = 0;
    int b = 1;
    cout << a << " " << b << endl;
    myswap(a, b);
    cout << a << " " << b << endl;

    return 0;
}
```

```
void myswap(int *&x, int *&y)
{
    int *t;
    t = x;
    x = y;
    y = t;
}

int main()
{
    int a = 0;
    int b = 1;
    int* p = &a;
    int* q = &b;
    cout << *p << " " << *q << endl;
    myswap(p, q);
    cout << *p << " " << *q << endl;

    return 0;
}
```

用对一维数组求和的方式对二维数组求和

原理：a[0]是第一行，求到末尾之后，下一个就是第二行的第一个元素，所以直接求和

```
int getSum(int a[], int num)
{
    int sum = 0;
    for (int i = 0; i < num; i++)
    {
        sum += a[i];
    }
    return sum;
}


int main()
{
    int a[2][3] = { {1,2,3},{4,5,6} };
    cout << getSum(a[0], 2 * 3) << endl;
    return 0;
}
```

没有自定义拷贝构造函数可能出现的问题：

```
class A
{       int x,y;
        char *p;
    public:
        A(char *str)
        {   x = 0;   y = 0;
            p = new char[strlen(str)+1];
            strcpy(p,str);
        }
        ~A() { delete [] p; p=NULL; }

};
.....
A a1("abcd");
A a2(a1);
```

存在问题：

A的隐式拷贝构造函数让a1和a2的p指针都指向了同一块内存，因此，a1对它的修改也会影响到a2；而且当a1与a2中一个消亡时，会归还p所指的空间，另一个a2的p就变成野指针了

解决方法：显式定义一个拷贝构造函数

```
A(const A &a)
{
    x = a.x;
    y = a.y;
    p = new char[strlen(a.p)+1]
    strcpy(p, a.p)
}
```

A	B	C	D	E	F	G	H	I	J	K	L
0	0	0	0	0	0	0	0	0	0	0	0
-1	-1	-1	1	1	0	0	0	0	1	0	0
3	-1	3	1	4	1	0	-1	1	1	0	0
5	-1	3	4	4	1	0	3	4	4	-1	0