首先，说起多态就必须要讲静态联编，动态联编。这俩也叫静态绑定和动态绑定。有些书比如C++ Primer也叫静态类型和动态类型。谭浩强写的C++程序设计直接叫静态多态性和动态多态性。

为什么说起多态就要先说他俩呢，首先，多态是什么？

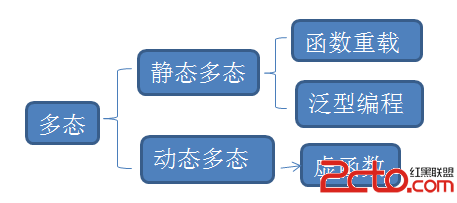
多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。引用Charlie Calverts对多态的描述——多态性是允许你将父对象设置成为一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作（摘自“Delphi4 编程技术内幕”）。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。多态性在Object Pascal和C++中都是通过虚函数（Virtual Function） 实现的。

上面的一段话讲得十分官方（哈哈），让我总结来说，多态就是一个事物有多种形态。

一、静态联编，动态联编，静态类型，动态类型

1.静态多态，动态多态

静态多态和动态多态的区别其实用下面的图就可以体现：



2.静态联编，动态联编

那么静态联编和动态联编分别是什么呢

首先我们先搞清楚联编是什么：

联编的作用是：程序调用函数，编译器决定使用哪个可执行代码块。

也就是确定调用的具体对象。

<code class="hljs cs">class A

{

public:

void Fun();

};

class B:public A

{

public:

void Fun();

};

int main()

{

B b;

b.Fun();

//上一行究竟调用A类的Fun（）函数还是B类的Fun函数

//确定具体对象的过程叫做联编

return 0;

//这个例子只是让大家了解一下什么是联编

//关于这个例子中涉及的知识点，在后面会提及

}</code>

正如上面所说的，联编的分类是根据进行阶段不同分类的。

静态联编其实就是类似上面我们提到的，函数重载和运算符重载，它是在编译过程汇总进行的联编，又称早期联编。

而动态联编是在程序运行过程中才动态的确定操作对象。

3.静态类型，动态类型

在C++Primer一书中，讲到了静态类型和动态类型。

静态类型和动态类型可用于变量或表达式。

表达式的静态类型在编译时总是已知的，它是在变量声明时的类型或表达式生成的类型。

动态类型则是变量或表达式表示的内存中的对象的类型，直到运行时才可知。

其实静态类型和动态类型与静态联编，动态联编是与指针和引用有着很大关系的。

原因如下：

实际上一个非指针非引用的变量，在声明时已经确定了它自己的类型，不会再后面改变。而指针或引用可以进行类型转换的原因，就是下面要好好分析的。

下面我要问一些问题。

1.什么有两种类型的联编？

2.既然动态联编如此好，为什么不将他设置成默认的？

3.动态联编是如何工作的？

现在我对上面的问题解答一下。

为什么有两种类型的联编以及为什么默认为静态联编？

原因有两个——效率和概念模型。

1.效率

为了使程序能够在运行阶段进行决策，必须采取一些方法来跟踪基类指针或引用指向的对象类型，这增加了额外的处理开秀，所以，在派生类不需要重新定义基类方法的情况下，静态联编的效率更高。

2.内存和存取时间，这点在后面虚函数的介绍中会提及

二、指针和引用类型兼容性

在我以前的博文C++继承详解之二——派生类成员函数详解（函数隐藏、构造函数与兼容覆盖规则）中，我提到过赋值兼容覆盖规则，其实就是这里的指针和引用类型兼容性。

在C++中，动态联编与指针和引用调用的方法相关。其实从某种程度上说，这是由继承控制的。在公有继承中，建立is-a关系的一种方法是如何处理指向对象的指针和引用。

一般情况下，C++是不允许将一种类型的地址赋给另一种类型的指针，也不允许一种类型的引用指向另一种类型。

<code class="hljs cs">float f;

int &a = f; //编译器报错

double \*p = &f;//编译器报错</code>

但是，在赋值兼容转换规则中，指向基类的引用或指针可以引用派生类对象，而不必进行显示类型转换。

?

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

<code class="hljs cs">

class Base

{

public:

int b;

};

class Derive:public Base

{

public:

int d;

};

int main()

{

Base b;

Derive d;

Base \*pb = &b;//基类指针指向基类对象

pb = &d;//基类指针指向派生类对象

Base &B = b;//基类引用指向基类对象

Base &D = d;//基类引用指向派生类对象

//编译运行都不会出错

return 0;

}</code>

**1.向上强制转换**

将派生类引用或指针转换为基类引用或指针被称为向上强制转换（upcasting），这使得公有继承不需要进行显示类型转换。这也是is-a规则的一部分（ C++继承详解之四——is-a接口继承和has-a实现继承）。

因为公有继承中是接口继承，即基类中的成员派生类中都有，所以发生向上强制转换的时候，势必担心出现问题的。

将指向对象对象的指针作为函数参数时，也是如此。

class Base

{

public:

int b;

};

class Derive:public Base

{

public:

int d;

};

int main()

{

Base b;

Derive d;

Base \*pb = &d;//向上强制转换

pb->b = 1; //可以赋值

cout << "Base::b = " << pb->b << endl;

return 0;

}

运行结果为：

**向上强制转换是可以传递的。**

即：我在派生类基础上再派生一个类，这时依然可以向上强制转换。< class Base { public: int b; }; class Derive:public Base { public: int d; }; class DerivePlus :public Derive { public: int dp; }; int main() { Base b; Derive d; DerivePlus dp; Base \*pb = &dp;//向上强制转换 pb->b = 1; //可以赋值 cout << "Base::b = " << pb->b << endl; return 0; }

运行结果为：

**2.向下强制转换**

与向上强制转换相反，将基类指针或引用转换为派生类指针或引用成为向下强制转换。

如果不使用显示类型转换，向下强制转换是不允许的，因为is-a关系是不可逆的。

比如香蕉是水果，但是水果不是香蕉。

派生类香蕉可以新增数据成员，因此这些数据成员不能应用于基类水果，比如香蕉中有黄色，但是不是所有水果都是黄色的。

<code class="hljs cs">class Fruit

{

public:

int weight;

};

class Banana:public Fruit

{

public:

int yellow;

};

int main()

{

Fruit b;

Banana d;

Banana \*pb = &b;//隐式向下强制转换 //报错

Banana \*p = (Banana\*)&b;//显式类型转换，不会报错

return 0;

}</code>

编译后编译器报错：

class="hljs livecodeserver">Error 1 error C2440: 'initializing' : cannot convert from 'Fruit \*' to 'Banana \*' e:\demo\继承\blog\project1\project1\source.cpp 93 1 Project1

3 IntelliSense: a value of type "Fruit \*" cannot be used to initialize an entity of type "Banana \*" e:\DEMO\继承\blog\Project1\Project1\Source.cpp 93 15 Project1

现在已经了解了向上向下强制转换是什么了。

那么隐式向上强制转换在使用的过程中是不会报错的，所以它是基类指针或引用可以指向基类对象以及派生类对象。因此需要动态联编。

C++使用虚成员函数来满足这种需求。

那么，什么是虚函数呢？

**三、虚函数**

1.什么是虚函数

虚函数就是在某基类中声明为 virtual 并在一个或多个派生类中被重新定义的成员函数。

用法格式为：virtual 函数返回类型 函数名（参数表） {函数体}；

实现多态性，通过指向派生类的基类指针或引用，访问派生类中同名覆盖成员函数。

如果没有使用关键字virtual,程序将根据引用类型或指针类型选择方法。

请看下面的例子：

<code class="hljs cpp">class Base

{

public:

int b;

void Fun()

{

cout << "Base::Fun()" << endl;

}

};

class Derive:public Base

{

public:

int d;

void Fun()

{

cout << "Derive::Fun()" << endl;

}

};

int main()

{

Base b;

Derive d;

Base \*pb = &d;//向上强制转换

pb->Fun();

return 0;

}</code>

上面的代码会输出什么呢？

上面的程序调用的Fun（）函数是基类的成员函数，也就是上面我所说的，程序是根据指针类型选择的方法，在上面的代码中，指针变量pb的类型是Base类，所以程序调用的是基类的成员函数Fun()。

那么我们加上virtual关键字，将Fun（）函数变为虚函数又会输出什么呢？

<code class="hljs cpp">class Base

{

public:

int b;

virtual void Fun()

{

cout << "Base::Fun()" << endl;

}

};

class Derive:public Base

{

public:

int d;

void Fun()

{

cout << "Derive::Fun()" << endl;

}

};

int main()

{

Base b;

Derive d;

Base \*pb = &d;//向上强制转换

pb->Fun();

return 0;

}</code>

上面的代码运行结果为:

这里写图片描述

那么可以得到，如果使用了virtual，程序将根据引用或指针指向的对象类型来选择方法。

在上面的例子中，虽然指针变量pb的类型为Base类，但是通过

<code class="hljs fix">Base \*pb = d;</code>

这句话，我们进行了向上强制转换，也就是说，Base类型的指针变量pb，指向了Derive类的对象，那么，在程序运行的时候，因为加了virtual关键字，Fun()函数变为了虚函数，所以程序将根据指针所指向的对象的类型，在本例中也就是Derive类型来选择函数。

如果将指针变为引用，结果类似。

<code class="hljs cpp">class Base

{

public:

int b;

virtual void Fun()

{

cout << "Base::Fun()" << endl;

}

};

class Derive:public Base

{

public:

int d;

void Fun()

{

cout << "Derive::Fun()" << endl;

}

};

int main()

{

Base b;

Derive d;

Base &pb = d;//向上强制转换

pb.Fun();

return 0;

}</code>

结果仍然为：

在后面将会看到，这种行为给我们带来了很大的方便。

要说明的是：

如果在基类中定义了虚函数，那么派生类中的同名函数将自动变为虚函数，但是我们可以在派生类同名函数前也加上virtual关键字，这样会增加程序的可读性。

总结：

如果要在派生类中重新定义基类的方法，通常应将基类方法声明为虚拟的。这样程序将根据对象类型而不是引用或指针的类型来选择方法也就是函数的版本

注意

这里一定要注意什么时候用虚函数，必须是用指针或引用调用方法的时候用虚函数，因为如果是通过对象调用方法，那么编译的时候就知道应该用哪个方法了。

2.虚函数的工作原理

C++规定了虚函数的行为，但将实现方法留给了编译器。

其实我们不需要知道实现方法就可以使用虚函数，但了解虚函数的工作原理会更好的帮助我们理解后面的更难的知识点，下面我就来剖析一下虚函数的工作原理。

通常，编译器处理虚函数的方法是：

给每个对象添加一个隐藏成员

隐藏成员中保存了一个指向函数地址数组的指针。

其实这里的函数地址数组指的就是虚函数表（virtual function table),vtbl。

虚函数表中存储了为类对象进行声明的虚函数的地址。

例如，基类对象包含一个指针，该指针指向基类中所有虚函数的地址表。派生类对象将包含一个指向独立地址表的指针。

如果派生类提供了虚函数的新定义，该虚函数表将保存新函数的地址，如果派生类没有重新定义虚函数，该vtbl将保存函数原始版本的地址。

如果派生类定义了新的虚函数，则该函数的地址也将被添加到vtbl中，注意，无论类中包含的虚函数是一个还是是个，都只需要在对象中添加一个地址成员，只是表的大小不同。

下面我将举一个例子并画出虚函数机制内存布局：

?

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

<code class="hljs cpp">class Base

{

public:

int b;

virtual void Fun1()

{

cout << "Base::Fun1()" << endl;

}

virtual void Fun2()

{

cout << "Base::Fun2()" << endl;

}

};

class Derive: public Base

{

public:

int d;

};

int main()

{

Base b;

b.b = 1;

Derive d;

d.b = 1;

d.d = 2;

return 0;

}</code>

看上面的例子，这个例子中派生类没有重新定义函数也没有新增函数。

我们看一下内存布局。