声明：现大部分文章为寻找问题时在网上相互转载，在此博客中做个记录，方便自己也方便有类似问题的朋友，故原出处已不好查到，如有侵权，请发邮件表明文章和原出处地址，我一定在文章中注明。谢谢。

董俊杰

xd502djj@163.com---识大势，懂取舍！分主次，懂先后！

开放、透明、反思。

业务思维、结果导向、成本意识。

< 2010年9月 >

日	一	二	三	四	五	六
29	30	31	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	28	29	30	1	2
3	4	5	6	7	8	9

- 如果一件事情你觉得难的完不成，你可以把它分为若干步,并不断寻找合适的方法。最后你会发现你是个超人。不要给自己找麻烦，但遇到麻烦绝不怕，更不要退缩。
- 电工查找电路不通点的最快方法是：分段诊断排除，快速定位。你有什么启示吗?
- 求知若饥，虚心若愚。
- 当你对一个事情掌控不足的时候，你需要做的就是"梳理"，并制定相应的规章制度，并使资源各司其职。
- 官网永远是获得第一时间获得第一手资料的最佳通道。
- 去繁归简:作为一个程序员，最痛恨的代码就是如老婆的裹脚布又臭又

C++ Virtual详解

Virtual是C++ OO机制中很重要的一个关键字。只要是学过C++的人都知道在类Base中加了Virtual关键字的函数就是虚拟函数（例如函数print），于是在Base的派生类Derived中就可以通过重写虚拟函数来实现对基类虚拟函数的覆盖。当基类Base的指针point指向派生类Derived的对象时，对point的print函数的调用实际上是调用了Derived的print函数而不是Base的print函数。这是面向对象中的多态性的体现。

（关于虚拟机制是如何实现的，参见Inside the C++ Object Model ，Addison Wesley 1996）

```
class Base
{
public:Base(){}
public:
      virtual void print(){cout<<"Base";}
};

class Derived:public Base
{
public:Derived(){}
```

长一样的代码；最崇尚的就是清晰、简洁、模块化的代码。

昵称：茄子_2008
园龄：9年9个月
粉丝：407
关注：11
+加关注

搜索

找找看

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

Linux(105)
Road(58)
Oracle(42)
Java(39)
Mysql(32)
C++(25)
Tools(25)
hadoop(21)
hive(18)
PHP(18)
更多

随笔档案

```
public:
    void print(){cout<<"Derived";}
};

int main()
{
    Base *point=new Derived();
    point->print();
}
```

Output:
Derived

这也许会使人联想到函数的重载，但稍加对比就会发现两者是完全不同的：

（1） 重载的几个函数必须在同一个类中；

覆盖的函数必须在有继承关系的不同的类中

（2） 覆盖的几个函数必须函数名、参数、返回值都相同；

重载的函数必须函数名相同，参数不同。参数不同的目的就是为了在函数调用的时候编译器能够通过参数来判断程序是在调用的哪个函数。这也就很自然地解释了为什么函数不能通过返回值不同来重载，因为程序在调用函数时很有可能不关心返回值，编译器就无法从代码中看出程序在调用的是哪个函数了。

（3） 覆盖的函数前必须加关键字Virtual；

重载和Virtual没有任何瓜葛，加不加都不影响重载的运作。

关于C++的隐藏规则：

我曾经听说过C++的隐藏规则：

（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

```
#include <iostream.h>
class Base
```

2018年9月 (2)
2018年8月 (5)
2018年7月 (9)
2018年6月 (3)
2018年5月 (2)
2018年4月 (5)
2018年3月 (2)
2018年1月 (4)
2017年12月 (3)
2017年11月 (5)
2017年10月 (2)
2017年9月 (4)
2017年8月 (6)
2017年7月 (6)
2017年6月 (13)
2017年5月 (3)
2017年4月 (6)
2017年3月 (9)
2017年2月 (6)
2017年1月 (4)
2016年12月 (2)
2016年11月 (7)
2016年10月 (4)
2016年9月 (1)
2016年8月 (1)
2016年7月 (1)
2016年6月 (5)
2016年5月 (2)
2016年3月 (1)
2016年2月 (1)
2016年1月 (1)

```
{
public:
virtual void f(float x){ cout << "Base::f(float) " << x << endl; }
void g(float x){ cout << "Base::g(float) " << x << endl; }
void h(float x){ cout << "Base::h(float) " << x << endl; }
};

class Derived : public Base
{
public:
virtual void f(float x){ cout << "Derived::f(float) " << x << endl; }
void g(int x){ cout << "Derived::g(int) " << x << endl; }
void h(float x){ cout << "Derived::h(float) " << x << endl; }
};

void main(void)
{
Derived d;
Base *pb = &d;
Derived *pd = &d;
// Good : behavior depends solely on type of the object
pb->f(3.14f); // Derived::f(float) 3.14
pd->f(3.14f); // Derived::f(float) 3.14
// Bad : behavior depends on type of the pointer
pb->g(3.14f); // Base::g(float) 3.14
pd->g(3.14f); // Derived::g(int) 3 (surprise!)
// Bad : behavior depends on type of the pointer
pb->h(3.14f); // Base::h(float) 3.14 (surprise!)
pd->h(3.14f); // Derived::h(float) 3.14
```

2015年12月 (3)
2015年11月 (1)
2015年10月 (3)
2015年9月 (6)
2015年8月 (2)
2015年7月 (19)
2015年6月 (2)
2015年5月 (3)
2015年4月 (8)
2015年3月 (7)
2015年2月 (8)
2015年1月 (6)
2014年12月 (6)
2014年11月 (18)
2014年10月 (1)
2014年9月 (3)
2014年8月 (8)
2014年7月 (1)
2014年6月 (3)
2014年5月 (2)
2014年4月 (1)
2014年3月 (3)
2014年2月 (4)
2014年1月 (3)
2013年12月 (7)
2013年11月 (4)
2013年10月 (2)
2013年9月 (8)
2013年8月 (10)
2013年7月 (5)
2013年5月 (6)
2013年4月 (2)

```
}
```

bp 和dp 指向同一地址，按理说运行结果应该是相同的，而事实上运行结果不同，所以他把原因归结为C++的隐藏规则，其实这一观点是错的。决定bp和dp调用函数运行结果的不是他们指向的地址，而是他们的指针类型。“只有在通过基类指针或引用间接指向派生类子类型时多态性才会起作用”(C++ Primer 3rd Edition)。pb是基类指针，pd是派生类指针，pd的所有函数调用都只是调用自己的函数，和多态性无关，所以pd的所有函数调用的结果都输出Derived::是完全正常的；pb的函数调用如果有virtual则根据多态性调用派生类的，如果没有virtual则是正常的静态函数调用，还是调用基类的，所以有virtual的f函数调用输出Derived::，其它两个没有virtual则还是输出Base::很正常啊，nothing surprise!
所以并没有所谓的隐藏规则，虽然《高质量C++/C 编程指南》是本很不错的书，可大家不要迷信哦。记住“只有在通过基类指针或引用间接指向派生类子类型时多态性才会起作用”。

纯虚函数:
C++语言为我们提供了一种语法结构，通过它可以指明，一个虚拟函数只是提供了一个可被子类型改写的接口。但是，它本身并不能通过虚拟机制被调用。这就是纯虚拟函数（pure virtual function）。 纯虚拟函数的声明如下所示:

```
class Query {
public:
// 声明纯虚拟函数
virtual ostream& print( ostream&=cout ) const = 0;
// ...
};
```

这里函数声明后面紧跟赋值0。
包含（或继承）一个或多个纯虚拟函数的类被编译器识别为抽象基类。试图创建一个抽象基类的独立类对象会导致编译时刻错误。（类似地通过虚拟机制调用纯虚拟函数也是错误的例如)

```
// Query 声明了纯虚拟函数
// 所以, 程序员不能创建独立的 Query 类对象
// ok: NameQuery 中的 Query 子对象
Query *pq = new NameQuery( "Nostromo" );
```

2013年3月 (4)
2013年2月 (6)
2013年1月 (8)
2012年12月 (1)
2012年11月 (5)
2012年10月 (2)
2012年9月 (5)
2012年8月 (1)
2012年7月 (4)
2012年6月 (7)
2012年5月 (1)
2012年4月 (2)
2012年3月 (5)
2012年1月 (1)
2011年12月 (3)
2011年11月 (8)
2011年10月 (1)
2011年9月 (2)
2011年8月 (4)
2011年7月 (10)
2011年6月 (3)
2011年5月 (2)
2011年4月 (2)
2011年3月 (13)
2011年2月 (4)
2011年1月 (4)
2010年12月 (4)
2010年11月 (8)
2010年10月 (2)
2010年9月 (42)
2010年8月 (18)
2010年7月 (5)

// 错误: new 表达式分配 Query 对象
Query *pq2 = new Query;
抽象基类只能作为子对象出现在后续的派生类中。

如果只知道virtual加在函数前，那对virtual只了解了一半，virtual还有一个重要用法是virtual public，就是虚拟继承。虚拟继承在C++ Primer中有详细的描述，下面稍作修改的阐释一下：
在缺省情况下C++中的继承是"按值组合"的一种特殊情况。当我们写
class Bear : public ZooAnimal { ... };
每个Bear 类对象都含有其ZooAnimal 基类子对象的所有非静态数据成员以及在Bear中声明的非静态数据成员类似地当派生类自己也作为一个基类对象时如：
class PolarBear : public Bear { ... };
则PolarBear 类对象含有在PolarBear 中声明的所有非静态数据成员以及其Bear 子对象的所有非静态数据成员和ZooAnimal 子对象的所有非静态数据成员。在单继承下这种由继承支持的特殊形式的按值组合提供了最有效的最紧凑的对象表示。在多继承下当一个基类在派生层次中出现多次时就会有问题最主要的实际例子是iostream 类层次结构。ostream 和istream 类都从抽象ios 基类派生而来，而iostream 类又是从ostream 和istream 派生
class iostream :public istream, public ostream { ... };
缺省情况下，每个iostream 类对象含有两个ios 子对象：在istream 子对象中的实例以及在ostream 子对象中的实例。这为什么不好？从效率上而言，存储ios 子对象的两个复本，浪费了存储区，因为iostream 只需要一个实例。而且，ios 构造函数被调用了两次每个子对象一次。更严重的问题是由于两个实例引起的二义性。例如，任何未限定修饰地访问ios 的成员都将导致编译时刻错误：到底访问哪个实例？如果ostream 和istream 对其ios 子对象的初始化稍稍不同，会怎样呢？怎样通过iostream 类保证这一对ios 值的一致性？在缺省的按值组合机制下，真的没有好办法可以保证这一点。
C++语言的解决方案是，提供另一种可替代按"引用组合"的继承机制虚拟继承（virtual inheritance ）在虚拟继承下只有一个共享的基类子对象被继承而无论该基类在派生层次
中出现多少次共享的基类子对象被称为虚拟基类。

通过用关键字virtual 修改一个基类的声明可以将它指定为被虚拟派生。例如，下列声明使得ZooAnimal 成为Bear 和Raccoon 的虚拟基类：
// 关键字 public 和 virtual

2010年6月 (5)
2010年5月 (1)
2010年4月 (3)
2010年3月 (2)
2009年10月 (1)
2009年7月 (2)
2009年6月 (6)
2009年3月 (5)
2009年2月 (7)

最新评论

1. Re:C++ Virtual详解

mark

--牛逼闪闪亮晶晶

2. Re:C++中的指针与引用

写的很好

--鄙视哥_小白

3. Re:Linux下如何查看系统启动时间和运行时间

@季成威多谢噢 上面加的有链接噢：）...

--茄子_2008

4. Re:Linux下如何查看系统启动时间和运行时间

部分原创

--季成威

5. Re:Linux下如何查看系统启动时间和运行时间

原创在这里

--季成威

```
// 的顺序不重要
class Bear : public virtual ZooAnimal { ... };
class Raccoon : virtual public ZooAnimal { ... };
```

虚拟派生不是基类本身的一个显式特性，而是它与派生类的关系如前面所说明的，虚拟继承提供了"按引用组合"。也就是说，对于子对象及其非静态成员的访问是间接进行的。这使得在多继承情况下，把多个虚拟基类子对象组合成派生类中的一个共享实例，从而提供了必要的灵活性。同时，即使一个基类是虚拟的，我们仍然可以通过该基类类型的指针或引用，来操纵派生类的对象。

标签: C++

好文要顶 关注我 收藏该文

茄子_2008
关注 - 11
粉丝 - 407
+加关注

9 0

« 上一篇: [C/C++]C++下基本类型所占位数和取值范围
» 下一篇: C++中的指针与引用

posted on 2010-09-22 02:39 茄子_2008 阅读(110237) 评论(5) 编辑 收藏

发表评论

#1楼 2013-03-17 21:50 | stushl 回复 引用

Derived类有没有继承 Base类的
void g(float x){ cout << "Base::g(float) " << x << endl; }
有的话pd->g(3.14f);为什么不调用这个？而非要调用那个接受int型参数的那个？所以我认为隐藏这一特性还是存在的。

支持(1) 反对(0)

#2楼 2014-08-05 13:37 | qin2013 回复 引用

mark

支持(2) 反对(0)

#3楼 2015-07-14 11:41 | sunyunzhuo 回复 引用

C++真是混乱。。。。。。。。。。。。。。。。。。。

支持(1) 反对(0)

#4楼 2015-12-09 17:28 | poisson_notes 回复 引用

@ qin2013
要知道float是可以隐式转换成int型的

支持(0) 反对(0)

#5楼 2018-02-25 22:41 | 牛逼闪闪亮晶晶 回复 引用

mark

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

发表评论

昵称: Justry2015

评论内容:

提交评论 退出 订阅评论

阅读排行榜

1. Linux查看CPU和内存使用情况(998714)
2. Linux 的cp命令(562301)
3. Linux添加/删除用户和用户组(516621)
4. java取整和java四舍五入方法(209789)
5. linux配置IP的方法(154716)

评论排行榜

1. Linux添加/删除用户和用户组(9)
2. Linux查看CPU和内存使用情况(9)
3. Linux下如何查看系统启动时间和运行时间(7)
4. C++ Virtual详解(5)
5. __declspec(dllexport) & __declspec(dllimport)(4)

推荐排行榜

1. Linux查看CPU和内存使用情况(22)
2. Linux添加/删除用户和用户组(19)
3. Linux 的cp命令(11)
4. C++ Virtual详解(9)
5. 恢复oracle中误删除drop掉的表(6)