**使用C语言的struct来实现C++的class**

2017年01月21日 17:42:52 [stophin](https://me.csdn.net/stophin) 阅读数：4620 标签： [struct](http://so.csdn.net/so/search/s.do?q=struct&t=blog) [class](http://so.csdn.net/so/search/s.do?q=class&t=blog) [gcc](http://so.csdn.net/so/search/s.do?q=gcc&t=blog) [模板](http://so.csdn.net/so/search/s.do?q=模板&t=blog) 更多

个人分类： [操作系统](https://blog.csdn.net/stophin/article/category/1714395)

如何在C语言中实现C++的Class呢?  
一些低级设备比如嵌入式，或者一些底层驱动、操作系统中，是不能使用C++语言的。网上有很多对这方面的解释，除非对编译器做一些设置或者修改，但这样大大增加了开发的难度。并且靠修改编译器参数来编译，仍旧不能利用到C++的优点。比如C++的虚函数，垃圾回收，异常，在底层开发中使用，反而会造成很多不必要的麻烦。比如C++编译器为了重载函数，其编译出来的函数名会被改成包括参数的形式，而且每个编译器都有自己的内部前缀和后缀，这一点尤其在操作系统编写中会造成麻烦，因为操作系统的系统调用使用汇编，比如中断的实现中，就需要调用汇编中断服务，然后在其中回调操作系统内核的C函数，如果使用C++函数，就不能正确指定回调函数名。那么在只能使用C语言来实现的情况下，如何让这种过程式语言更具封装性，而不让代码看起来“懒散”呢？  
C语言中可以和class类比的类型就是struct了，另外还有union， 但union并不具备class的条件。在struct中不能定义函数， 这一点可以在Microsoft Visual Studio中和Linux GCC下做个比较：  
typedef struct A {  
int data;  
int Val() {  
 return data;  
}  
}A;  
A a;  
a.Val();  
在VS下这个struct能通过编译，并且a.Val()能取到值， 这是因为C++编译器在对兼容C语言的struct进行编译时，是将struct按照public class来理解的，所以能支持内联函数。但GCC是只支持C语言的编译器，编译时就会报错。那么，如果使用C语言，如何才能让struct媲美class呢？  
其实C类语言都支持函数指针的定义，并且struct中也支持函数指针定义。比如  
int func(int a, int b);  
定义的函数指针可以使这样的：  
int (\*pfunc)(int, int);  
当定义pfunc = func时，下面两个调用是一样的：  
func(10, 20);和pfunc(10, 20);  
那如上面所说，将函数指针定义到struct中：  
typedef struct A {  
int data;  
int (\*Val)(int a);  
}A;  
int Val(int a) {  
 return a;  
}  
A a;  
a.Val = Val;  
a.Val(10);  
这样可以得到10的结果。我们知道class中隐含了一个this指针，那在Val函数中怎样才能得到this呢？对了，就通过参数传递进去：  
typedef struct A  A;  
struct A{  
int data;  
int (\*Val)(A\* that, int a);  
};  
int Val(A\* that, int a) {  
 return that->data + a;  
}  
A a;  
a.Val = Val;  
a.Val(&a, 10);  
上面就可以得到a.data + 10的结果。我们使用that来代替this，这样如果这段代码拿到C++编译器下面时也不会跟struct中隐含的this冲突。这样就定义了struct来代替class，唯一的缺点是定义好以后，每次调用函数需要将对象指针传递进去，这也是无可避免的。可以定义下面的宏来防止两次指定的对象不一致：  
#define \_\_CALL(o, f, ...) o.f(&o, \_\_VA\_ARGS\_\_)  
\_\_CALL(a, Val, 10);  
其中\_\_VA\_ARGS\_\_是C语言关键字，用于将宏的变参传递到指定位置。在编译期宏就已经被展开，因此Val已经是a的成员（函数）了，所以不用担心Val这个参数在\_\_CALL这个宏调用时没有定义。

**进阶1： 构造函数**上一步中，a.Val = Val;写在外面，如果有好几个成员（函数），会造成代码臃肿，需要定义一个构造函数。  
A \* \_A(A\* that, int data) {  
that->data = data;  
that->Val = Val;  
return that;  
}  
A a;  
\_A(&a, 20);  
a.Val(&a, 10);  
这样定义一个对象就需要两行代码，仍旧可以定义宏来实现新建对象，不过如果是new对象完全没有必要。  
A \* a = \_A(new A, 10);  
另外这里构造函数只能是一个普通函数，不能作为成员（函数），因为构造函数只调用一次，没有作为成员的必要，并且构造函数如果是成员也没法在构造前知道构造函数是什么，因此只能在外部指定。如果非要定义成成员（函数），这就变成了两行代码：  
a.init = \_A;  
a.init(&a, 20);  
哈哈，但是如果想要重新设置对象a的值， 定义init成员则另当别论，不过最好还是在普通函数\_A中定义。

**进阶2：继承**我们已经有了一个很好的"class"了：  
typedef struct A  A;  
struct A{  
int data;  
int (\*Val)(A\* that, int a);  
};  
int Val(A\* that, int a) {  
 return that->data + a;  
}  
A \* \_A(A \* that, int data) {  
that->data = data;  
that->Val = Val;  
return that;  
}  
A a;  
\_A(&a, 20);  
a.Val(&a, 10);  
接下来，我们要实现继承。因为如果只是需要上面的代码，就没有使用类的必要，实现继承才是使用类的终极目标。这里先暂时、而且也没法实现虚函数之类的，不用考虑这些。实现继承需要用到上面提到的union，比如继承上面的A:  
typedef struct B B;  
struct B {  
union {  
A super;  
struct {  
int data;  
int (\*Val)(A\* that, int a);  
};  
};  
int val;  
};  
B\* \_B(B\* that, int val) {  
\_A(&that->super val);  
that->val = val;  
}  
B b;  
\_B(&b, 30);  
在union中，定义了基类A，以及将基类A中的成员都拷贝到了一个匿名struct中。在C规范中，union的匿名struct后面定义一个变量名那么使用变量名.成员才能得到变量，而如果没有变量名，则直接使用成员名就能得到变量，如下：  
union {  
float f[2];  
struct {  
float f1;  
float f2;  
}uf;  
}um;  
要得到f[1]使用um.fu.f2可以得到，而  
union {  
float f[2];  
struct {  
float f1;  
float f2;  
};  
}um;  
只使用um.f2就能得到f[1]。我们的类就是利用了这点，可以让基类的成员变成继承类的成员。继承类中super部分是基类，而自身又定义了val这个成员变量，是属于基类以外的，而且更有意思的是，在B的构造函数中，可以直接通过that->super来构造a，并且构造函数完了以后，b.data和b.Val就是构造A以后的成员，它们分别等于b.super.data和b.super.Val。

**进阶3:部分继承和重写（not重载）**另外我们还可以使用这种结构来实现部分继承。来看下面A和B的定义（只有定义，前向声明和调用省略）：  
struct A{  
int data;  
int (\*Val)(A\* that, int a);  
int val;  
};  
struct B {  
union {  
A super;  
struct {  
int data;  
int (\*Val)(A\* that, int a);  
};  
};  
int val;  
};  
其中，B.val和A.val是不同的成员。如果要取得A.val使用b.super.val就可以了， 这个是union的特性来决定的。这种没有继承基类的情况叫部分继承。  
那么怎么实现重写呢？来看B的构造函数：  
B\* \_B(B\* that, int val) {  
\_A(&that->super, val);  
//override  
that->Val = myVal;  
that->val = val;  
}  
只要将继承类的Val指针指向自定义的函数就可以了。不过注意必须在A构造完成之后，否则会被覆盖回来。  
所以归纳起来，构造函数的写法顺序为：  
基类构造  
重写函数  
子类构造（函数指针初始化）  
子类数据初始化  
其他初始化  
其中，“其他初始化”之前的所有都可以类比为C++类的构造函数：  
B : A(val), val(val) {}  
而重载函数是在C++类中定义内联函数时就已经直接重载了。

**进阶4：定义宏使结构更简单**C语言模拟C++类的大体如上。不过如果想要创建类更简便，还需要一些宏定义的帮助。  
在任何情况下，我们都应该知道自己定义这个类将来是不是会成为基类。所以，对于A，我们知道它是基类，可以将它改写成“模板”，但这个模板非C++的template，只是宏定义，用于简化继承类中基类的书写：  
typedef struct A {  
#define A\_Template \  
int data;\  
int (\*Val)(A\* that, int a);  
#define Template\_A A\_Template  
Template\_A  
int val;  
}A;  
虽然多了两个#define，但是A的定义并没有变化。int val;不是由子类继承的所以不用写在#define里面。#define包括在花括号内是为了让代码更加美观。#define将会在下面宏中使用：  
#define \_\_SUPER(Base) \  
union {\  
 Base super; \  
 struct {\  
  Template\_##Base\  
  }; \  
 }  
这就是B的union部分，我们将它提炼出来，使得以后所有的类都可以不失一般性地调用宏来继承。  
typedef struct B {  
\_\_SUPER(A);  
int val;  
}B;  
看，这样就不用再去重新写基类的成员了，所有基类成员只在基类中定义一遍，在子类中通过宏来进行展开。

**进阶5：模板？(simulate template)**我们通过上面的说明，能够很快写出一个类的例子，这是一个能编译运行的C代码(linux gcc)：  
// Class.c  
//  
#include <stdio.h>  
#include <stdlib.h>  
  
//////////////////////////////////////////////////////  
#define \_\_SUPER(Base) \  
union {\  
 Base super; \  
 struct {\  
  Template\_##Base\  
  }; \  
 }  
//////////////////////////////////////////////////////  
typedef struct A A;  
struct A {  
#define A\_Template \  
int data;\  
int (\*Val)(A\* that, int a);  
#define Template\_A A\_Template  
Template\_A  
int val;  
};  
int Val(A\* that, int a) {  
 return that->data + a;  
}  
A \* \_A(A \* that, int data) {  
that->data = data;  
that->Val = Val;  
return that;  
}  
//////////////////////////////////////////////////////  
typedef struct B {  
\_\_SUPER(A);  
int val;  
}B;  
int myVal(B\* that, int a) {  
 return that->data \* a;  
}  
B\* \_B(B\* that, int val) {  
\_A(&that->super, val);  
that->Val = myVal;  
that->val = val;  
}  
//////////////////////////////////////////////////////  
int main() {  
 A a;  
 \_A(&a, 10);  
 a.val = 1;  
 printf("%d %d\n", a.val, a.Val(&a, 20));  
  
 B b;  
 \_B(&b, 20);  
 b.val = 2;  
 printf("%d %d\n", b.val, b.Val(&b, 30));  
   
 return 0;  
}  
可以看到结果是1 20和2 600，说明成功了。但是编译器报出警告 ：assignment from incompatible pointer type，这是因为基类Val是A类型参数，而B中重载时给的却是B类型参数，由于是继承关系，并且是指针，所以不用去理会也不会有什么问题。但是如果真的要较真的话，就需要更改#define了：  
struct A {  
#define A\_Template(T) \  
int data;\  
int (\*Val)(T\* that, int a);  
#define Template\_A(T) A\_Template(struct T)  
Template\_A(A)  
int val;  
};  
上面\_SUPER和下面B调用\_\_SUPER的地方也一并改掉，不赘述。可以看到，当在A中时，Val使用的A类型参数，而在B中时使用B类型参数，应该不会有问题了。--------但这并不是模板！因为struct的局限性，我们通过添加参数为函数传入that指针代替this指针，我们定义类型T是为了除去继承时指针类型不匹配的问题， 但并没有引入模板。  
我们在继承类中使用宏来展开基类，这一点跟模板很像，为什么不能做成模板呢？哈哈，我们可以模仿一下模板，但真正的模板并不是这样的：  
// Class.c  
//  
#include <stdio.h>  
#include <stdlib.h>  
  
//////////////////////////////////////////////////////  
#define \_\_SUPER(Base, Type, ...) \  
union {\  
 Base super; \  
 struct {\  
  Template\_##Base(Type, \_\_VA\_ARGS\_\_)\  
  }; \  
 }  
//////////////////////////////////////////////////////  
typedef struct A A;  
struct A {  
#define A\_Template(T, Type) \  
Type data;\  
int (\*Val)(T\* that, int a);  
#define Template\_A(T, Type) A\_Template(struct T, Type)  
Template\_A(A, int)  
int val;  
};  
int Val(A\* that, int a) {  
 return that->data + a;  
}  
A \* \_A(A \* that, int data) {  
that->data = data;  
that->Val = Val;  
return that;  
}  
//////////////////////////////////////////////////////  
typedef struct B {  
\_\_SUPER(A, B, int);  
int val;  
}B;  
int myVal(B\* that, int a) {  
 return that->data \* a;  
}  
B\* \_B(B\* that, int val) {  
\_A(&that->super, val);  
that->Val = myVal;  
that->val = val;  
}  
//////////////////////////////////////////////////////  
int main() {  
 A a;  
 \_A(&a, 10);  
 a.val = 1;  
 printf("%d %d\n", a.val, a.Val(&a, 20));  
  
 B b;  
 \_B(&b, 20);  
 b.val = 2;  
 printf("%d %d\n", b.val, b.Val(&b, 30));  
   
 return 0;  
}  
看，模板就是#define template那里，而A在定义的时候就已经将模板特化成int类型，B在定义时将模板特化成int类型。结果仍旧一样。但这个仍旧不是模板，模板是在定义对象的时候特化，而这个是在定义类型时已经特化。  
由于成员函数并不是内联的（所谓内联，就是说每一个对象包含的函数都是在对象内部扩展开的），所以这些函数必须写在外部，而外部必须保证struct进行了完全定义，所以C是没有办法做到真正的模板的。

**6.继承链**根据C的宏定义标准，宏是不可以嵌套的。因此如果要实现继承链，\_\_SUPER会造成嵌套。因此，再增加一个\_\_\_SUPER表示二级继承链，定义和\_\_SUPER一模一样，这样，\_\_\_SUPER会调用Template然后调用\_\_SUPER，构成继承链。**如果还有更多继承，则继续定义\_\_\_\_SUPER**。另外，由于使用的是匿名union，因此基类的super都暴露在外面，继承链中会出现重复super定义，可以将super定义为super+基类名来区别。

二级继承链定义如下，一级继承链也修改为下面的定义：  
#define \_\_\_SUPER(Base, ...)\  
union {\  
 Base super##Base; \  
 struct {\  
  Template\_##Base(\_\_VA\_ARGS\_\_)\  
  }; \  
 }  
好了，这样就可以实现多级继承链了。在\_B构造函数中将super改为superA，然后添加C类继承B类（二级继承）：  
struct C {  
\_\_\_SUPER(B, C);  
};  
C \* \_C(C\* that) {  
\_B(&that->superB, 77);  
}  
这样就构成了一个完整的继承链。完整可运行代码如下：  
// Class.c  
//  
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct A A;  
typedef struct B B;  
typedef struct C C;  
//////////////////////////////////////////////////////  
#define \_\_SUPER(Base, ...)\  
union {\  
 Base super##Base; \  
 struct {\  
  Template\_##Base(\_\_VA\_ARGS\_\_)\  
  }; \  
 }  
#define \_\_\_SUPER(Base, ...)\  
union {\  
 Base super##Base; \  
 struct {\  
  Template\_##Base(\_\_VA\_ARGS\_\_)\  
  }; \  
 }  
//////////////////////////////////////////////////////  
struct A {  
#define A\_Template(T) \  
int data;\  
int (\*Val)(T\* that, int a);  
#define Template\_A(T) A\_Template( T)  
Template\_A(A)  
int val;  
};  
int Val(A\* that, int a) {  
 return that->data + a;  
}  
A \* \_A(A \* that, int data) {  
that->data = data;  
that->Val = Val;  
return that;  
}  
//////////////////////////////////////////////////////  
struct B {  
#define B\_Template( T) \  
\_\_SUPER(A, T);\  
int val;  
#define Template\_B( T) B\_Template( T)  
Template\_B( B)  
};  
int myVal(B\* that, int a) {  
 return that->data \* a;  
}  
B\* \_B(B\* that, int val) {  
\_A(&that->superA, val);  
that->Val = myVal;  
that->val = val;  
}  
//////////////////////////////////////////////////////  
struct C {  
\_\_\_SUPER(B, C);  
};  
C \* \_C(C\* that) {  
\_B(&that->superB, 77);  
}  
//////////////////////////////////////////////////////  
int main() {  
 A a;  
 \_A(&a, 10);  
 a.val = 1;  
 printf("%d %d\n", a.val, a.Val(&a, 20));  
  
 B b;  
 \_B(&b, 20);  
 b.val = 2;  
 printf("%d %d\n", b.val, b.Val(&b, 30));  
   
 C c;  
 \_C(&c);  
 c.val = 3;  
 printf("%d %d\n", c.val, c.Val(&c, 30));  
   
 return 0;  
}  
输出结果：1 30 2 600 3 2310，结果正确。  
  
7.结束语  
虽然没能做到模板很遗憾，但是能够保证一些class的元素能够被使用在C语言中，已经很不错了。本次class的应用是在看操作系统内核时，寻找用C语言解决class问题时，其中的一些研究成果，并且在实践中已经证实可以使用----虽然有些晦涩而且并不是一个很好的编程体验（比如下面记载的，对于习惯了class的人来说真的很难改变）

但是需要强调几点，这也是遗留的问题，以现在的能力还是没有办法解决，期待高人来解决吧：  
1.使用和定义类成员（函数）时一定要带上类对象指针that以代替this  
2.struct的继承和C++的继承有很大区别，因为struct是通过在子类中重新定义基类来进行的，因此，基类设计一定要注意成员顺序，比如上面例子中A的val是放在最后而不是最前，这样通过子类继承时子类才不会将val继承过去。  
3.由于宏定义的一些特性，导致继承链需要定义多个级别的\_\_SUPER，需要明确当前继承是第几级，然后确定使用哪个\_\_SUPER  
4.第2点所带来的麻烦并不止这些，如果是连续继承，所有子类必须拥有基类的所有成员，继承链越靠后类就会变得越臃肿，这仅仅是指类所占的空间，然而在代码中使用\_\_SUPER并不会看到这种放大作用  
5.如上看到的，所有的成员函数并不是内联函数，而仅仅是指向函数的指针，指向了一个struct外部的函数（在gcc中是不允许直接将函数写在struct中的，因此只能定义函数指针，其实也相当于一个成员变量而已），所以成员函数没有办法做内联优化。  
完(\*^\_^\*)  
  
stophin 2016/11/26