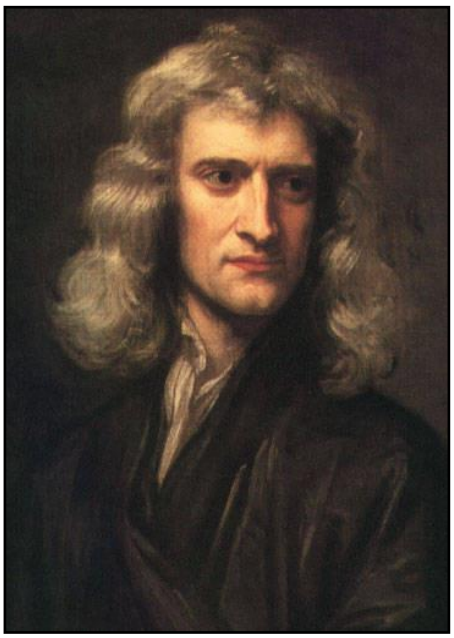
OopC

An OOP Library for C

Goodman Tao/陶兴

2019-09-27



无比崇敬的*Isaac Newton*爵士

目录

1. 起因

“OopC”是“Object Oriented Programming in C”的缩写。应该归咎于我的偏执吧，我特别喜欢 C 语言简约的语法和高效的性能，但很遗憾的是它没有面向对象的特性。我曾试图深入理解和学习 C++，但是它太过庞大，这显然和驻留在我头脑里的观点是不一致的。优雅的事物应该是简单而又强大的，或者说用相对简单的事物取得更大的效益，或者说我所认为的最优化。

事实上不乏前辈追求或追求过同样的事情。和他们的追求或许有些不同，我的另一个目标是实用，而不是完全空洞的优雅或者不切实际的追求性能，它们之间需要合理的权衡。这也在 OopC 的代码中有所体现——提供面向对象特性的支持，而非达到“所有的都是对象”的目的。

2. 源码

OopC 的源代码托管在 GitHub[1]和码云[2]上，编译环境为 Visual Studio 2017，使用标准 C99 实现。采用 MIT 授权，也就是说，如果我的代码有幸能供您使用，还望您将软件著作权声明的证书附加到所使用的项目中，此外，没有其它的要求了。如果能够通知我一下，您正在使用这些代码，我会非常高兴的。

代码的目录如图 2.1，

图 2.1

doc/ 包含项目的说明文档。OopC.sln 为项目解决方案。OopC/ 包含与项目核心库代码和相

[1] https://github.com/GoodmanTao/OopC.git
[2] https://gitee.com/goodmantao/OopC.git

关的示例工程，如下图 2.2，

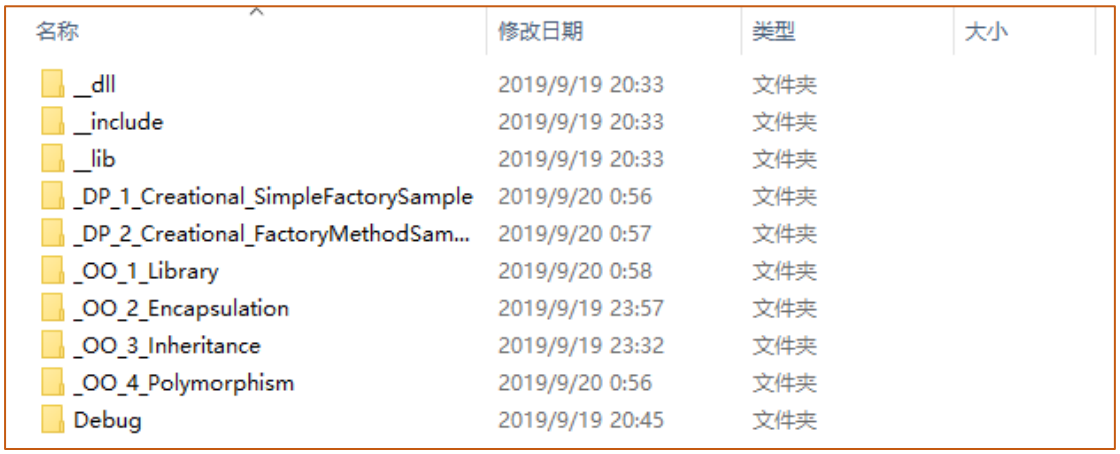


图 2.2

文件夹名称前缀“OO”表示“Object Oriented”。_OO_1_Library/ 包含 OopC 的核心代码，该工程在编译后输出 OopBase.dll 至__dll/，输出 OopBase.lib 至__lib/，复制 OopBase.h 至__include/。其余三个以“OO”开头的目录，正如其名称后缀一样，为示例代码，分别演示 OopC 的封装、继承和多态特性。“DP”前缀表示“Design Pattern”，以此前缀开头的文件夹表示该文件夹下的工程演示一种设计模式的实现，比如上图中“_DP_1_Creational_SimpleFactorySample”，该工程演示设计模式中的创建型模式，具体为简单工厂模式。到文档撰写日止，只完成了两个模式的示例代码的编写，后面会持续更新。文件夹中的数字表示推荐阅读顺序，个人推荐先按序阅读“OO”部分，然后阅读“DP”部分。

3. Hello, world!

照例，我们先体验一下 OopC 的“Hello, world!”，当然不是演示打印一条语句，这里通过该例子，展示 OopC 是如何支持 C 语言面向对象特性的。

- HelloWorld.h

```
#ifndef HELLOWORLD_H__
#define HELLOWORLD_H__
#include <OopBase.h>

CLASSDEF(HelloWorld);

typedef ParamNull HelloWorld_Print;

#endif // !HELLOWORLD_H__
```

- HelloWorld.c

```
#include "HelloWorld.h"

```

```
3  //#include <malloc.h>
4  #include <stdio.h>
5
6  struct HelloWorld
7  {
8      CHAINDEF;
9  };
10
11 ////////////////////////////////////////////////////////////
12 //
13
14 static void Print(void *pParams)
15 {
16     HelloWorld *pThis = ((ParamIn *)pParams)->pInst;
17     HelloWorld_Print *pIn = ((ParamIn*)pParams)->pIn;
18
19     printf("Hello, world!.\n");
20 }
21
22 ////////////////////////////////////////////////////////////
23 //
24
25 bool INVOKE(HelloWorld)
26 (HelloWorld *pInst, char *pFuncName, void *pParams)
27 {
28  DOINVOKE(pInst, pFuncName, pParams);
29 }
30
31 void *EXTEND(HelloWorld)(HelloWorld *pInst)
32 {
33  DOEXTEND(pInst);
34 }
35
36 void DELETE(HelloWorld)(HelloWorld **ppInst)
37 {
38     DODELETE(ppInst, HelloWorld, Object);
39 }
40
41 HelloWorld *CREATE(HelloWorld)()
42 {
43       DOCREATE(pCreate, HelloWorld, Object, NULL,
44       METHOD(Print));
45      return pCreate;
46 }
```

➢ main.c

```
1  #include "HelloWorld.h"
2
3  int main(int argc, char **argv)
4  {
5       HelloWorld *pHllWrld = CREATE(HelloWorld)();
6
7       INVOKE(HelloWorld)(pHllWrld, "Print", NULL);
8
9       DELETE(HelloWorld)(&pHllWrld);
10
11      return 0;
12 }
```

头文件 HelloWorld.h 中第 1、2、9 行的条件编译指令防止重复包含类。第 3 行的预编译包含指令包含 OopBase.h，这是 OopC 的核心头文件。第 5 行的 CLASSDEF 类函数宏入参为 HelloWorld，表示定义一个类 HelloWorld。第 7 行 typedef 类型定义的含义为，类 HelloWorld 有一个名为 Print 的方法，方法的输入和输出参数为空（使用 ParamNull 示意，定义该结构体的目的也在于此）。至此，HelloWorld 类的类定义就完成了。

类实现文件 HelloWorld.c 中，第 1 行包含类头文件。第 3、4 行包含类实现时所需的头文件。第 6~9 行包含类的数据域结构体，这个结构体的名称必须与类名相同。在 OopC 中，所有的类都应该有数据域结构体，每个数据域结构体的名称都应该与类名相同。每个类数据域结构体都应包含 CHAINDEF，如代码第 8 行所示。

第 14~20 行定义类的成员方法 Print，您可能会发现，在说明头文件 HelloWorld.h 时，曾提到 Print 方法输入输出参数为空，而这里的定义并非如此——Print 的输入参数为指针类型。实际上，在类定义时指明 Print 方法的输入输出参数为空，意在说明 pIn 参数为 NULL，如代码第 17 行所示。这里需要重点指出的是类的所有成员方法的存储类别都推荐声明为静态内部链接 static，入参都必须声明为空指针类型 void *，返回参数都推荐声明为空类型 void。第 16 行获取类的 this 指针，第 17 行获取输入输出参数，因为 Print 方法的输入输出参数为空，所以运行时 pIn 的值也就是 NULL。

第 25~46 行，也就是实现文件中的剩下部分内容，为类的控制函数，在 OopC 中，每个类都有 4 个控制函数：调用函数（INVOKE）、扩展函数（EXTEND）、删除函数（DELETE）、创建函数（CREATE），删除函数和创建函数也就是析构函数和构造函数。调用函数用于帮助客户代码调用实例的方法，在 main.c 中有其使用示例。扩展函数有两个作用，其一是帮助子类继承该类，其二是子类到父类的类型转换，如何使用，在继承示例_OO_3_Inheritance 中有所体现。所有类的调用函数和扩展函数都是一致的，删除函数也是大同小异，创建函数除了不同的类的具体业务逻辑之外，大部分代码都相同。第 38 行进行内存清除。第 43、44 行创建类实例，METHOD(Print)指明该方法可以被客户代码通过实例调用，如果还有一个方法 Dummy 可以让外界调用，第 43 行 NULL 后的参数则应改为 METHOD(Print)METHOD(Dummy)，在本例中，第 14~20 行实际已经给出了 Print 方法的实现，如果添加的方法没有实现，则应改为 AMETHOD(Print),A 的含义为 Abstract,AMETHOD 的在继承示例中有使用示例。第 43 行的 Object 参数说明类 HelloWorld 继承了它，NULL 对应的形参，用于释放类相关的额外内存，具体怎么使用在后续章节将会有具体示例予以说明，pCreate 指出类函数宏 DOCREATE 创建了一个名为

pCreate 的实例。

客户代码 main.c 中，第 5 行创建类实例，在 OopC 中，有创建则有删除，因此第 9 行删除类实例。第 7 行，与一般的函数调用形式相异，OopC 中对成员方法的调用通过类的控制函数——调用函数实现，类函数宏 INVOKE 的输入参数为类名称 HelloWorld，表示调用类 HelloWorld 的调用函数，以实现成员方法的调用，第一个参数为类实例，表示调用将调用该实例的成员方法，第二个参数为"Print"，表示希望调用名"Print"为的成员方法，第三个参数为 NULL，表示调用该方法的传入传出参数为空。

好了，现在我们的“Hello, world!”的例子就完成了。注意在编译的时候需要添加头文件 OopBase.h，库文件 OopBase.lib，运行时需要添加动态链接库文件 OopBase.dll。运行成功后，控制台将打印 Hello, world!。

4. Object

Object 类是 OopC 核心库中唯一的一个类，包含在 OopBase.h 中，在 OopC 中，每个类都是 Object 类的子类，这并不意味着您设计的类会自动继承 Object 类，相反如果您设计的类没有继承自其它类，则应该显式继承该类。

下面是 Object 类的定义，

```
1  typedef struct Object Object;
2
3  Object* CREATE(Object)();
4  bool  INVOKE(Object)(Object*   pInst, char* pFuncName, void* pParams);
5  void* EXTEND(Object)(Object*   pInst);
6  void  DELETE(Object)(Object** ppInst);
7
8  typedef struct { bool* pRet; void* pToCmpr; } Object_Equal;
9  typedef ParamNull Object_ToString;
```

OopC 中，类本质是结构体和函数的一个功能组合。结构体变量作为类实例，而函数因为只能通过结构体变量来调用所以成为类的成员方法。因此第 1 行将结构体声明为 Object 类型，也就声明了 Object 类。

在“Hello,world!”示例中，类头文件 HelloWorld.h 中并没有显示声明四个控制函数，这是因为在宏 CLASSDEF 中已经给出了声明，这里因为没有使用该宏，因此第 3~6 行显式声明四个控制函数：创建函数 CREATE(Object)、调用函数 INVOKE(Object)、扩展函数 EXTEND(Object)和删除函数 DELETE(Object)。第 3 行可以看出，创建函数没有输入参数，等价于无参构造函数。第 4 行调用函数包含三个输入参数：类实例、将要调用的方法名称字符串和该方法的输

入输出参数，它是一个结构体指针，什么样的结构体指针呢？不急。第 5 行为扩展函数，入参为类实例，返回一个指针。第 6 行为删除函数，它的参数为类实例的地址。

我们先看第 9 行，从“Hello，world!”示例我们可以知道，Object 类包含一个 ToString 方法，该方法的输入输出参数为 NULL。从第 8 行可以看出，Object 类还包含一个 Equal 方法，该方法和 ToString 不同，它有两个参数，bool 指针 pRet 为返回值（当然，您可以将它作为输入参数，不过这好像不太优雅），表明比较结果，void*指针 pToCmpr 为需要与之比较的参数。我们刚才说到，调用函数的第三个参数为一个结构体指针，现在应该比较清晰了，这个结构体就是对应方法的出入参数结构体。下面用一个例子来说明，

```
1  bool bRet = false;
2  Object *pToCmpr = CREATE(Object)();
3  Object *pObj = CREATE(Object)();
4  INVOKE(Object)(pObj, "Equal", &(Object_Equal){&bRet, pToCmpr});
```

5. OOP

5.1. 继承

在工程_OO_3_Inheritance 中，演示了 OopC 的继承。首先定义了一个计算器 Calculator 类，用于模拟算术运算功能，代码如下：

- Calculator.h

```
1  #ifndef CALCULATOR_H__
2  #define CALCULATOR_H__
3
4  #include <OopBase.h>
5
6  CLASSDEF(Calculator);
7
8  typedef struct { double dblOpL; double dblOpR; } Calculator_Input;
9  typedef ParamNull Calculator_Add;
10 typedef struct { double *pdblRet; } Calculator_Output;
11 typedef struct { double dblOpL; double dblOpR; double *pdblRet; }
12 Calculator_Subtract;
13 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
14 Calculator_Multiply;
15
16 #endif // !CALCULATOR_H__
```

第 8 行定义了一个出入参数结构体 Calculator_Input，表明该类具有一个名为 Input 的方法，该方法用于输入两个操作数 dblOpL 和 dblOpR 到计算器中。第 9 行表明类具有方法 Add，其输入输出参数为空，该方法执行求和操作。第 10 行表明通过调用类实例的 Output 方法可

以获取计算结果。第 11~14 行表明类还存在 Subtract 方法和 Multiply 方法，分别对应减法和乘法，dblOpL 和 dblOpR 均为输入操作数，dblOpRet 为计算得到的结果——差和乘积。观察可以发现，类 Calculator 所模拟的计算器有两种求值方式，加法为惰性求值，而减法和乘法为“即时求值”。容易理解，Add 方法惰性求值，所以需要存储两个输入参数和相应的输出结果。

下面来看计算器类 Calculator 的实现方式，代码如下：

- Calculator.c

```
1  #include "Calculator.h"
2
3  //#include <malloc.h>
4
5  struct Calculator
6  {
7        CHAINDEF;
8
9        double dblOperandL;
10       double dblOperandR;
11
12       double dblResult;
13 };
14
15 ////////////////////////////////////////////////////////////////
16 //
17
18 static void Input(void* pParams)
19 {
20       Calculator* pThis = ((ParamIn*)pParams)->pInst;
21       Calculator_Input* pIn = ((ParamIn*)pParams)->pIn;
22
23       pThis->dblOperandL = pIn->dblOpL;
24       pThis->dblOperandR = pIn->dblOpR;
25 }
26
27 static void Add(void* pParams)
28 {
29       Calculator* pThis = ((ParamIn*)pParams)->pInst;
30       Calculator_Add* pIn = ((ParamIn*)pParams)->pIn;
31
32       pThis->dblResult = pThis->dblOperandL + pThis->dblOperandR;
33 }
34
35 static void Output(void* pParams)
36 {
37       Calculator* pThis = ((ParamIn*)pParams)->pInst;
38       Calculator_Output* pIn = ((ParamIn*)pParams)->pIn;
39
40       *pIn->pdblRet = pThis->dblResult;
41 }
42
43 static void Subtract(void* pParams)
44 {
```

```
45        Calculator* pThis = ((ParamIn*)pParams)->pInst;
46        Calculator_Subtract* pIn = ((ParamIn*)pParams)->pIn;
47
48        *pIn->pdblRet = pIn->dblOpL - pIn->dblOpR;
49  }
50
51  static void Multiply(void* pParams)
52  {
53        Calculator* pThis = ((ParamIn*)pParams)->pInst;
54        Calculator_Multiply* pIn = ((ParamIn*)pParams)->pIn;
55
56        *pIn->pdblRet = pIn->dblOpL * pIn->dblOpR;
57  }
58
59  ////////////////////////////////////////////////////////////////////
60  //
61
62  bool INVOKE(Calculator)
63  (Calculator* pInst, char* pFuncName, void* pParams)
64  {
65        DOINVOKE(pInst, pFuncName, pParams);
66  }
67
68  void* EXTEND(Calculator)(Calculator* pInst)
69  {
70       DOEXTEND(pInst);
71  }
72
73  void DELETE(Calculator)(Calculator** ppInst)
74  {
75       DODELETE(ppInst, Calculator, Object);
76  }
77
78  Calculator* CREATE(Calculator)()
79  {
80       DOCREATE(pCreate, Calculator, Object, NULL,
81          METHOD(Input)
82          METHOD(Add)
83          METHOD(Output)
84          METHOD(Subtract)
85          METHOD(Multiply));
86
87        return pCreate;
88  }
```

第 9~12 行类的数据域中定义了三个变量，用于存储两个操作参数和返回值。第 59~88 行为类的控制函数。类成员 Input、Add、Output、Subtract 和 Multiply 方法的前两行获取 this 指针和方法的输入参数。整体的逻辑非常简单。

可以看出类 Calculator 作为算术运算模型，它缺少除法运算，我们将通过 OopC 的继承特性来扩展该类。另外，我们再提出一点新的需求，将加法改为“即时求值”。改造的后的代码如下，

- CalculatorExt.h

```
1  #ifndef CALCULATOREXT_H__
2  #define CALCULATOREXT_H__
3
4  #include "Calculator.h"
5
6  CLASSDEF(CalculatorExt);
7
8  typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
9  CalculatorExt_Add;
10 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
11 CalculatorExt_Subtract;
12 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
13 CalculatorExt_Multiply;
14 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
15 CalculatorExt_Divide;
16
17 #endif // !CALCULATOREXT_H__
```

第 4 行包含类 Calculator 的头文件，表明类 CalculatorExt 继承自 Calculator。第 8~15 行的出入参数结构体表明类具有加减乘除运算的方法，他们都具有两个操作数和一个返回值。代码 CalculatorExt.c 展示类 CalculatorExt 是如何继承和扩展类 Calculator 的。

- CalculatorExt.c

```
1  #include "CalculatorExt.h"
2
3  //#include <malloc.h>
5
6  struct CalculatorExt
7  {
8        CHAINDEF;
9  };
10
11 ////////////////////////////////////////////////////////////////////
12 //
13
14 static void Add(void* pParams)
15 {
16       CalculatorExt* pThis = ((ParamIn*)pParams)->pInst;
17       CalculatorExt_Add* pIn = ((ParamIn*)pParams)->pIn;
18
19       //这里类似于对象组合技术中的调用转发
20       DOINVOKESUPER(pThis, "Input",
21              &(Calculator_Input) { pIn->dblOpL, pIn->dblOpR });
22       DOINVOKESUPER(pThis, "Add", NULL);
23       DOINVOKESUPER(pThis, "Output",
24              &(Calculator_Output){pIn->pdblRet});
25 }
26
27 //static void Subtract(void* pParams);//直接继承
28
29 static void Multiply(void* pParams)
30 {
31       CalculatorExt* pThis = ((ParamIn*)pParams)->pInst;
32       CalculatorExt_Multiply* pIn = ((ParamIn*)pParams)->pIn;
33
34       //调用转发
```

```
35        DOINVOKESUPER(pThis, "Multiply", pIn);
36  }
37
38  static void Divide(void* pParams)
39  {
40        CalculatorExt* pThis = ((ParamIn*)pParams)->pInst;
41        CalculatorExt_Divide* pIn = ((ParamIn*)pParams)->pIn;
42
43        *pIn->pdblRet = pIn->dblOpL / pIn->dblOpR;//除0.。。
44  }
45
46  ////////////////////////////////////////////////////////////////
47  //
48
49  bool INVOKE(CalculatorExt)
50  (CalculatorExt* pInst, char* pFuncName, void* pParams)
51  {
52        DOINVOKE(pInst, pFuncName, pParams);
53  }
54
55  void* EXTEND(CalculatorExt)(CalculatorExt* pInst)
56  {
57       DOEXTEND(pInst);
58  }
59
60  void DELETE(CalculatorExt)(CalculatorExt** ppInst)
61  {
62       DODELETE(ppInst, CalculatorExt, Calculator);
63  }
64
65  CalculatorExt* CREATE(CalculatorExt)()
66  {
67       DOCREATE(pCreate, CalculatorExt, Calculator, NULL,
68          METHOD(Add)
69          METHOD(Multiply)
70          METHOD(Divide));
71
72        return pCreate;
73  }
```

为了将 Add 方法扩展成“即时求值”的计算方式，我们在第 14~25 行重写了 Add 方法，该方法前两行获取 this 指针和输入参数。第 20~24 行调用了三次类函数宏 DOINVOKESUPER，使用该宏可以调用父类的方法，譬如这里依次调用了父类 Calculator 的三个方法 Input、Add 和 Output 实现求和运算，在 C 语言中，实现两个数相加的功能自然是非常简单的，这里没有直接将两个输入参数 pIn->dblOpL 和 pIn->dblOpR 相加然后返回，而是将求和请求转发至父类，这样做的目的是想说明 OopC 中的一种通过转发调用实现代码复用的方式。实际上，子类并不能访问父类的私有方法[3]和所有数据域，子类和其它使用父类的客户代码一样，都只能访问父类所暴露出来的接口，这里父类 Calculator 所暴露出来的接口只有 Input、Add、

[3] 这里的私有方法指的是，在父类构造方法 DOCREATE 中没有作为 METHOD 入参的方法。

Output、Subtract 和 Multiply。在父类中已经存在“即时求值”的 Subtract 方法，通过继承，子类获取到父类的该方法，因此在第 27 行通过注释予以说明，无需再次实现该方法，同样不出您所料，子类也自动继承了 Input 和 Output 方法。第 29~36 行以调用转发的方式重写了父类的 Multiply 方法，这里旨在说明当父类指针引用子类实例时，如果通过该指针调用 Multiply 方法，则子类的同名方法将被调用。第 38~44 行扩展了计算器的除法计算功能，除 0 的问题没有解决，不过这个问题现在不是我们关注的重点。第 46~73 行的代码实现类 CalculatorExt 的控制函数，其中第 68~70 行，Divide 方法为新增加的方法以实现除法功能，自然要作为 DOCREATE 的参数以声明其为公共方法，需要注意的是 Add 和 Multiply 方法，因为它们被重写，所以需要添加到 DOCREATE 的参数中。

5.2. 多态

面向对象技术中的多态是指父类型根据实际类型产生不同的行为的特性，它与继承是紧密相关的。示例代码中，工程_00_4_Polymorphism 演示了 OopC 的多态特性。该示例代码首先定义了一个计算器基类 CalculatorBase，然后两个子类 CalculatorExtdCommon 和 CalculatorExtdEnhanced 分别继承该类，从而扩展出一般的计算器和加强版的计算器。

- CalculatorBase.h

```
1  #ifndef CALCULATORBASE_H__
2  #define CALCULATORBASE_H__
3
4  #include <OopBase.h>
5
6  CLASSDEF(CalculatorBase);
7
8  typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
9  CalculatorBase_Add;
10 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
11 CalculatorBase_Subtract;
12 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
13 CalculatorBase_Multiply;
14 typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
15 CalculatorBase_Divide;
16
17 #endif // !CALCULATORBASE_H__
```

类 CalculatorBase 定义了 4 个算术运算方法，出入参数分别包含两个操作数和一个计算结果。

- CalculatorBase.c

```
1  #include "CalculatorBase.h"
2
3  //#include <malloc.h>
```

```
4
5  struct CalculatorBase
6  {
7         CHAINDEF;
8  };
9
10 ////////////////////////////////////////////////////////////
11 //
12
13 ////////////////////////////////////////////////////////////
14 //
15
16 bool INVOKE(CalculatorBase)
17 (CalculatorBase* pInst, char* pFuncName, void* pParams)
18 {
19        DOINVOKE(pInst, pFuncName, pParams);
20 }
21
22 void* EXTEND(CalculatorBase)(CalculatorBase* pInst)
23 {
24       DOEXTEND(pInst);
25 }
26
27 void DELETE(CalculatorBase)(CalculatorBase** ppInst)
28 {
29       DODELETE(ppInst, CalculatorBase, Object);
30 }
31
32 CalculatorBase* CREATE(CalculatorBase)()
33 {
34       DOCREATE(pCreate, CalculatorBase, Object, NULL,
35           AMETHOD(Add)
36           AMETHOD(Subtract)
37           AMETHOD(Multiply)
38           AMETHOD(Divide));
39
40        return pCreate;
41 }
```

在类 CalculatorBase 的声明中有 4 个算术运算方法，但是在类的实现中并没有给出定义，这意味着，该类实例化后事实上无法调用对应的方法进行计算，这当然是由我们设定的业务逻辑决定的，该示例中，我们只需要通过父类引用子类即可，所以仅仅在父类构造时，第 35~38 行，声明了该类包含 4 个用于算术运算的成员方法。需要注意的是，在讲解 OopC 的继承特性时，类函数宏 DOCREATE 添加方法使用的是 METHOD，而这里使用的是 AMETHOD，这样做的目的是表明所添加的方法没有实现，前缀 A 的含义是 Abstract。

两个子类 CalculatorExtdCommon 和 CalculatorExtdEnhanced 继承基类，它们的实现是比较简单的，最主要的就是实现父类的同名成员方法，然后在创建函数中通过宏 DOCREATE 添加这 4 个方法。下面只给出类 CalculatorExtdCommon 的代码，加强版计算器类 CalculatorExtdEnhanced 的代码与之区别不大，

- CalculatorExtCommon.h

```
#ifndef CALCULATOREXTDCOMMON_H__
#define CALCULATOREXTDCOMMON_H__

#include "CalculatorBase.h"

CLASSDEF(CalculatorExtdCommon);

typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
CalculatorExtdCommon_Add;
typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
CalculatorExtdCommon_Subtract;
typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
CalculatorExtdCommon_Multiply;
typedef struct { double dblOpL; double dblOpR; double* pdblRet; }
CalculatorExtdCommon_Divide;

#endif // !CALCULATOREXTDCOMMON_H__
```

- CalculatorExtCommon.c

```
#include "CalculatorExtdCommon.h"

#include <stdio.h>
//#include <malloc.h>

struct CalculatorExtdCommon
{
	CHAINDEF;
};

////////////////////////////////////////////////////////////////////
//

static void Add(void* pParams)
{
	CalculatorExtdCommon* pThis = ((ParamIn*)pParams)->pInst;
	CalculatorExtdCommon_Add* pIn = ((ParamIn*)pParams)->pIn;

     printf("Add operation using common calculator.\n");

	*pIn->pdblRet = pIn->dblOpL + pIn->dblOpR;
}

static void Subtract(void* pParams)
{
	CalculatorExtdCommon* pThis = ((ParamIn*)pParams)->pInst;
	CalculatorExtdCommon_Subtract* pIn = ((ParamIn*)pParams)->pIn;

     printf("Subtract operation using common calculator.\n");

	*pIn->pdblRet = pIn->dblOpL - pIn->dblOpR;
}

static void Multiply(void* pParams)
{
	CalculatorExtdCommon* pThis = ((ParamIn*)pParams)->pInst;
	CalculatorExtdCommon_Multiply* pIn = ((ParamIn*)pParams)->pIn;
```

```
38
39      printf("Multiply operation using common calculator.\n");
40
41       *pIn->pdblRet = pIn->dblOpL * pIn->dblOpR;
42 }
43
44 static void Divide(void* pParams)
45 {
46       CalculatorExtdCommon* pThis = ((ParamIn*)pParams)->pInst;
47       CalculatorExtdCommon_Divide* pIn = ((ParamIn*)pParams)->pIn;
48
49      printf("Divide operation using common calculator.\n");
50
51       *pIn->pdblRet = pIn->dblOpL / pIn->dblOpR;// */0
52 }
53
54 ////////////////////////////////////////////////////////////////////
55 //
56
57 bool INVOKE(CalculatorExtdCommon)
58 (CalculatorExtdCommon* pInst, char* pFuncName, void* pParams)
59 {
60       DOINVOKE(pInst, pFuncName, pParams);
61 }
62
63 void* EXTEND(CalculatorExtdCommon)(CalculatorExtdCommon* pInst)
64 {
65      DOEXTEND(pInst);
66 }
67
68 void DELETE(CalculatorExtdCommon)(CalculatorExtdCommon** ppInst)
69 {
70      DODELETE(ppInst, CalculatorExtdCommon, CalculatorBase);
71 }
72
73 CalculatorExtdCommon* CREATE(CalculatorExtdCommon)()
74 {
75      DOCREATE(pCreate, CalculatorExtdCommon, CalculatorBase, NULL,
76         METHOD(Add)
77         METHOD(Subtract)
78         METHOD(Multiply)
79         METHOD(Divide));
80
81       return pCreate;
82 }
```

6. 特性

6.1. 构造函数

“Talk is cheap, show me the code”。OopC 中类的创建（构造）函数的特性通过下

面的例子来说明应该更加清晰明了，

```
1  #ifndef CALCULATOR_H__
2  #define CALCULATOR_H__
3
4  #include <OopBase.h>
5
6  CLASSDEF(Calculator, double dblOperandL, double dblOperandR);
7
8  typedef struct { double dblOperandL; double dblOperandR; } Calculator_Input;
9  typedef ParamNull Calculator_Add;
10 typedef struct { double* pResult; } Calculator_Output;
11
12 #endif // !CALCULATOR_H__
```

前面章节的示例中，我们已经多次见到过类的声明方式，形如 `CLASSDEF(Actor)`，第 6 行代码所表露出的内容有些不同，后面增加了两个 `double` 类型的形参，**这样声明表示类 `Calculator` 的创建函数含有两个 `double` 类型的参数，同时也意味着类 `Calculator` 将没有无参创建函数**。这样声明以后，类 `Calculator` 在实现时，其创建函数类似如下代码，

```
1  Calculator *CREATE(Calculator)(double dblOperandL, double dblOperandR)
2  {
3    //...
4  }
```

客户代码创建实例则类似如下代码，

```
1  Calculator *pCalculator = CREATE(Calculator)(1.0, 2.1);
```

6.2. 调用函数

6.3. 类额外内存释放

什么样的内存是类额外内存？动态分配给类的数据域字段的内存就是类额外内存。比如下面是类定义的部分代码，

```
-  #ifndef PERSON_H__
-  #define PERSON_H__
-
-  #include <OopBase.h>
-
6  CLASSDEF(Person, char *pName);
-
-  ... ...
-
-  #endif // !PERSON_H__
```

该类声明的第 6 行可以知道，需要一个名字作为构造参数，理所应当，在类的内部也应该分配相应的内存来存放实例的名字，其类实现的部分代码如下，

```
- #include "Person.h"
-
- #include <stdio.h>
```

```
- ... ...
- struct Person
- {
-    CHAINDEF;
9    char *pName;
- };
- ... ...
```

数据域声明了存储名字的指针后(第 9 行)，在创建（构造）函数中，也就可以通过下面的方式，保存通过创建函数形参传入的名字了，如下面代码第 21~22 行。另外，我们再分散一点注意力，阅读一下第 17 行的代码和注释。

```
- Person* CREATE(Person)(char *pName)
- {
17   DOCREATE(pCreate, Person, Object, ?,//注意这里的问号，我们在此立一个flag
-         ... ...); //这里的参数与非本节的关注点，故省略号替代
-
20   pCreate->pName = NULL;
21   int nLen = strlen(pName) + 1;
22   pCreate->pName = memcpy(realloc(pCreate->pName, nLen), pName, nLen);
-    ... ...
-    return pCreate;
- }
```

在创建 Person 实例时分配了内存存放其名字，后面就是使用它了。实际情况下，并非所有人一生都只使用一个名字，有可能中途更改名字。因此 Person 类应该给出更改名称的公共方法，如果新改的名字比原来的名字要长，那么就需要重新分配内存，并且需要释放存放原有名字的内存，具体视业务逻辑而定。无论如何，我们在使用名字字符串的时候，都应该保证两点：1.为它分配新的内存的时候，原有内存应该手动释放；2.名字指针要么为 NULL，要么引用一段动态分配的内存。满足这两点意味着，不再使用 Person 实例的时候，始终存在一段内存，也就是名字指针引用的内存，需要释放。OopC 中，针对这一类需要释放的内存设置了额外内存释放机制。

首先，定义一个函数用于释放额外内存，该函数的声明类型是固定的，返回值必须为 void，入参只能有一个，且必须为空指针类型，比如释放 Person 类的名字对用的内存，示例代码如下，

```
- static void ClearExtraMem(void *pToClear)
- {
-    Person *pInst = (Person *)pToClear;
-    free(pInst->pName);
- }
```

可以看出，在函数中调用了 free 释放内存，那怎样调用呢？我们只需要构造一个释放额外内存专用的结构体，然后传入到问号（“？”，还记得第 17 行代码的的 flag 吗，返回去看看），剩下的，当我们调用类的删除（析构）函数 DELETE(Person)(Person** ppInst)时，这段额外内存就会被随之释放。

最后一个问题，怎样构造这个结构体呢，请看如下代码，

```
-    ... ...
17   DOCREATE(pCreate, Person, Object, GenerateExtraMemRef(ClearExtraMem, pCreate),
-         ... ...);
-    ... ...
```

使用 OopC 库给出的接口函数 GenerateExtraMemRef(ExtraMemClear fnExec, void* pToClear)，以释放额外内存的函数和类实例指针为参数，返回值即为所需的结构体指针，很简单吧。

7. 规范

8. API 汇总

9. 原理

10.设计模式实例

11.下一步

12.一起

如果您喜欢 OopC，并且想加入进来贡献自己的智慧，请记住，我一直在等待您这样的朋友。如果您在阅读代码后，有什么更好的建议，真诚的希望您能够向我提出来。如果使用过程中有什么问题，记得加 QQ 群咨询哦，我们的 QQ 据点是 530484560。

13.感谢

OopC 是我非常喜欢的，应该说是一件作品吧。工作之余也投入了许多精力去思考和实践。如果对您有丝毫帮助，我将非常高兴。

谢谢您的捐赠！

14.^_^


C语言OOP库一一(
扫一扫二维码，加入群
推荐使用微信支付
(*兴)
微信支付
支付宝
ALIPAY
支付就用支付宝
(*兴)
打开支付宝[扫一扫]
免费寄送收钱码：拨打95188-6