

面向对象编程在 C 语言中的应用

艾恩凝

#公众号：技术乱舞

博客：<https://aeneag.xyz>

Application NoteObject-Oriented Programming in C

艾恩凝

<https://aeneag.xyz>

#公众号：技术乱舞

声明：本文章翻译于外网文档，翻译内容未经授权禁止复制转载

文章于2021/12/29翻译

面向对象编程在C语言中的应用

Application NoteObject-Oriented Programming in C

面向对象编程在C语言中的应用

0.介绍

1. 封装

2. 继承

3. 多态性（虚拟函数）

3.1虚拟表（vtbl）和虚拟指针（vptr）

3.2 在构造函数中设置vptr

3.3 继承vtbl并重写子类中的vptr

3.4 虚拟调用（后期绑定）

3.5 使用虚函数的示例

4. 总结

5. 参考文献

法律免责声明

本文件中的信息被认为是准确可靠的。但是，Quantum Leaps 不对此类信息的准确性或完整性做出任何明示或暗示的陈述或保证，并且对使用此类信息的后果不承担任何责任。

Quantum Leaps 保留随时更改本文件中发布的信息的权利，包括但不限于规范和产品说明，恕不另行通知。本文件取代并替换本文件出版前提供的所有信息。

0. 介绍

面向对象编程（OOP）不是使用特定的语言或工具。这是一种基于三个基本概念的设计方式：

- 封装–将数据和函数打包到类中的能力
- 继承–基于现有类定义新类的能力，以便获得重用和代码组织
- 多态性–在运行时将匹配接口的对象相互替换的能力

虽然这些设计模式传统上与面向对象语言（例如SimulaTalk、C++或java）相关联，但是您可以在几乎任何编程语言中实现它们，包括便携的、标准兼容的C语言。

注意：

如果你只在C中开发最终用户程序，但是你也想要做OOP，你可能应该使用C++而不是C。相比于C++，C中的OOP可能是繁琐的和容易出错的，并且很少提供任何性能优势。

但是，如果您构建软件库或框架，OOP概念可以作为组织代码的主要机制非常有用。在这种情况下，在C中执行OOP的大部分困难可以局限于库中，并且可以有效地对应用程序开发人员隐藏。本文档考虑了这个主要用例。

本应用说明描述了如何在QP/C和QP nano实时框架中实现OOP。作为这些框架的用户，您需要了解这些技术，因为您还需要将它们应用到您自己的应用程序级代码中。但这些技术不仅限于开发QP/C或QP纳米应用程序，而且通常适用于任何C程序。

代码下载：可以从Github [OOP-in-C](#) 下载本应用程序说明附带的代码。

译者提供代码链接-----

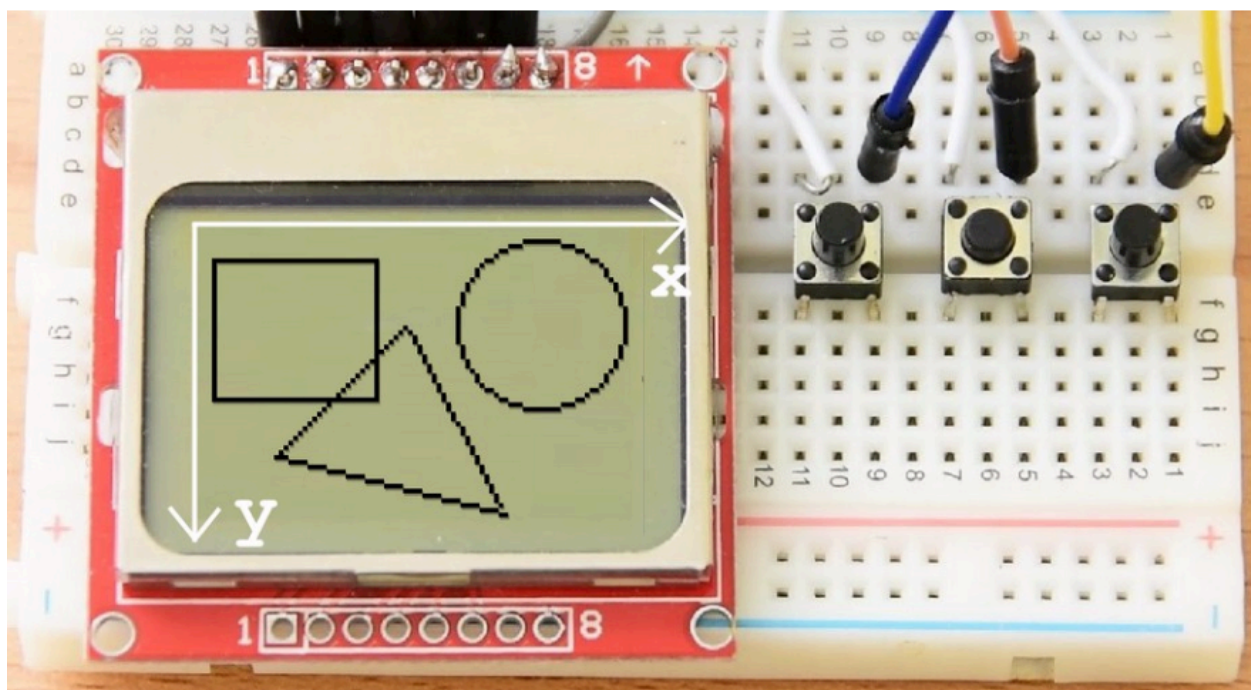
1. 封装

封装是将带有函数的数据打包到类中的能力。实际上，任何C程序员都应该非常熟悉这个概念，因为即使在传统的C语言中也经常使用它。例如，在标准C运行时库中，函数族包括 `fopen ()`、`fclose ()`、`fread ()`、`fwrite ()` 等。

对FILE类型的对象进行操作。因此，FILE结构被封装，因为客户端程序员不需要访问FILE结构的内部属性，相反，文件的整个接口只包含上述函数。您可以将FILE结构和在其上操作的相关C函数视为FILE class。以下项目概括了C运行时库如何实现FILE class：

1. 类的属性是用C结构（FILE struct）定义的。
2. 类的操作定义为C函数。每个函数都将指向属性结构（FILE *）的指针作为参数。类操作通常遵循通用的命名约定（例如，所有FILE类方法都以前缀f开头）。
3. 特殊函数初始化和清理属性结构（fopen（）和fclose（））。这些函数分别扮演类构造函数和析构函数的角色。

你可以很容易地应用这些设计原则来创建自己的“类”。例如，假设一个使用二维几何形状的应用程序（可能要在嵌入式图形LCD上渲染）。



C实现的shae class可以声明如下：

```
1  #ifndef SHAPE_H
2  #define SHAPE_H
3
4  #include <stdint.h>
5
6  // Shape 的属性
7  typedef struct {
8      int16_t x;
```

```

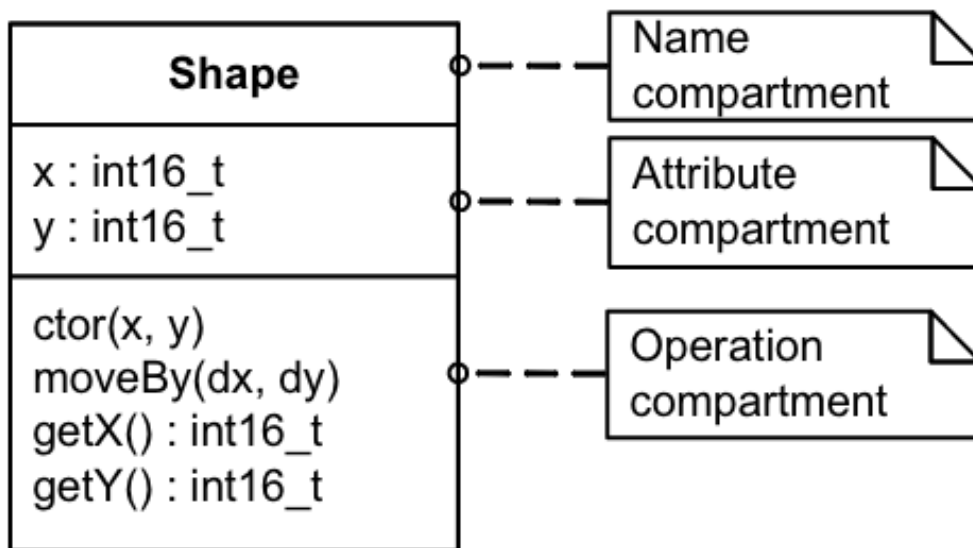
9     int16_t y;
10 } Shape;
11
12 // Shape 的操作函数，接口函数
13 void Shape_ctor(Shape * const me, int16_t x, int16_t y);
14 void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);
15 int16_t Shape_getX(Shape const * const me);
16 int16_t Shape_getY(Shape const * const me);
17
18 #endif /* SHAPE_H */

```

Shapeclass声明通常放在头文件（例如Shape.h）中，尽管有时您可能会选择将声明放在.c文件中。

类的一个很好的方面是可以在图表中绘制它们，图表显示类名、属性、操作和类之间的关系。下图显示了Shape类的UML类图：

Figure 1 UML Class Diagram of the Shape class



下面是形状操作的定义（必须在.c文件中）：

```

1 #include "shape.h"/* Shape class interface */
2
3 /* constructor */
4 void Shape_ctor(Shape * const me, int16_t x, int16_t y)

```

```
5 {
6     me->x = x;
7     me->y = y;
8 }
9 /* move-by operation implementation */
10 void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy)
11 {
12     me->x += dx;
13     me->y += dy;
14 }
15
16 // 获取属性值函数
17 int16_t Shape_getX(Shape const * const me)
18 {
19     return me->x;
20 }
21 int16_t Shape_getY(Shape const * const me)
22 {
23     return me->y;
24 }
```

你可以创建任意数量的shape object作为shape struct的实例。您需要使用“构造函数 Shape_ctor()初始化每个实例。您只能通过提供的操作来操作形状，这些操作将指针“me”作为第一个参数。

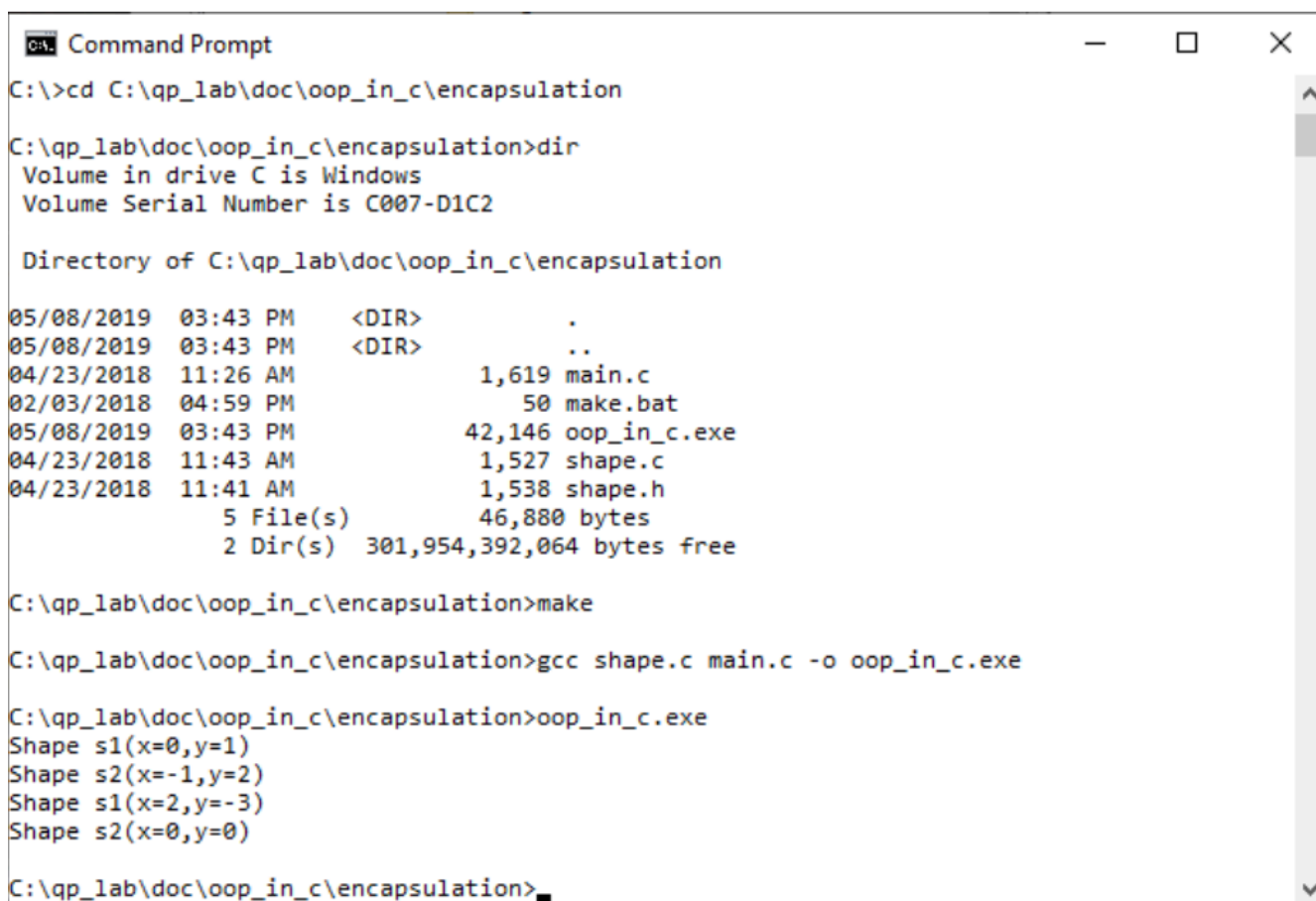
注意：代码中的me就是c++中的this

main函数如下：

```
1 #include "shape.h" /* Shape class interface */
2 #include <stdio.h> /* for printf() */
3
4 int main()
5 {
6     Shape s1, s2; /* multiple instances of Shape */
7
8     Shape_ctor(&s1, 0, 1);
9     Shape_ctor(&s2, -1, 2);
10 }
```

```
11     printf("Shape s1(x=%d,y=%d)\n", Shape_getX(&s1), Shape_getY(&s1));
12     printf("Shape s2(x=%d,y=%d)\n", Shape_getX(&s2), Shape_getY(&s2));
13
14     Shape_moveBy(&s1, 2, -4);
15     Shape_moveBy(&s2, 1, -2);
16
17     printf("Shape s1(x=%d,y=%d)\n", Shape_getX(&s1), Shape_getY(&s1));
18     printf("Shape s2(x=%d,y=%d)\n", Shape_getX(&s2), Shape_getY(&s2));
19
20     return 0;
21 }
```

运行结果：



```
C:\>cd C:\qp_lab\doc\oop_in_c\encapsulation

C:\qp_lab\doc\oop_in_c\encapsulation>dir
Volume in drive C is Windows
Volume Serial Number is C007-D1C2

Directory of C:\qp_lab\doc\oop_in_c\encapsulation

05/08/2019  03:43 PM    <DIR>          .
05/08/2019  03:43 PM    <DIR>          ..
04/23/2018  11:26 AM                1,619 main.c
02/03/2018  04:59 PM                 50 make.bat
05/08/2019  03:43 PM            42,146 oop_in_c.exe
04/23/2018  11:43 AM             1,527 shape.c
04/23/2018  11:41 AM             1,538 shape.h
               5 File(s)            46,880 bytes
               2 Dir(s)  301,954,392,064 bytes free

C:\qp_lab\doc\oop_in_c\encapsulation>make

C:\qp_lab\doc\oop_in_c\encapsulation>gcc shape.c main.c -o oop_in_c.exe

C:\qp_lab\doc\oop_in_c\encapsulation>oop_in_c.exe
Shape s1(x=0,y=1)
Shape s2(x=-1,y=2)
Shape s1(x=2,y=-3)
Shape s2(x=0,y=0)

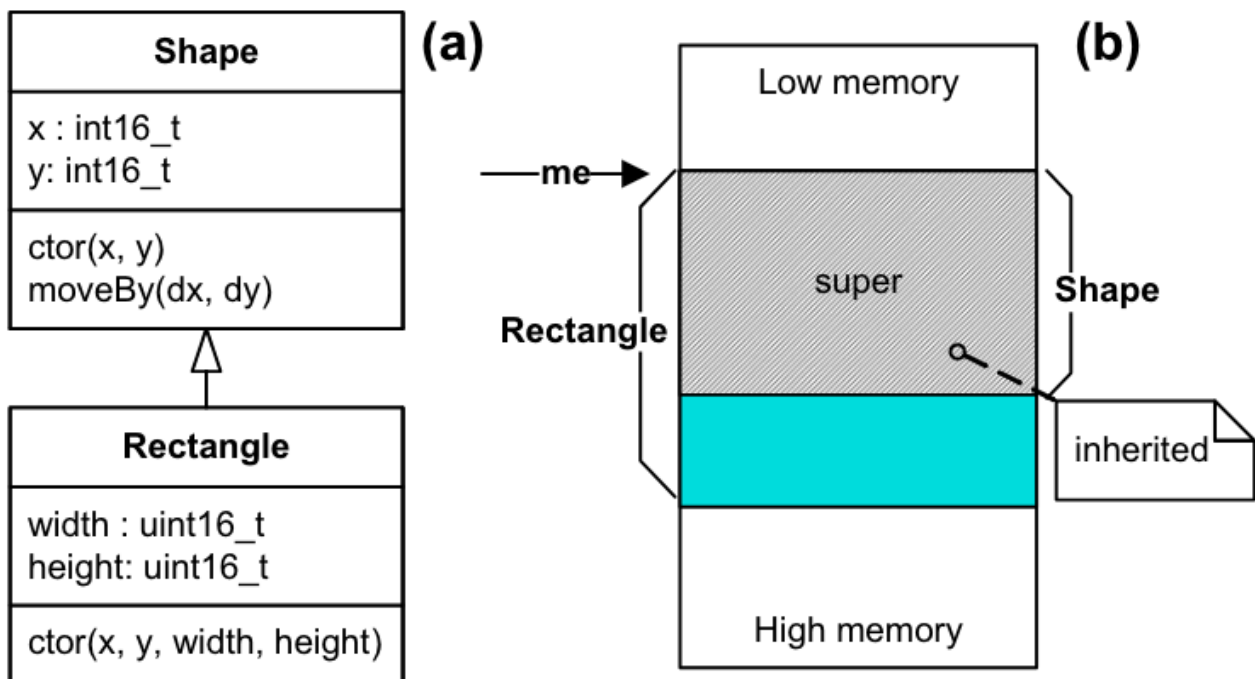
C:\qp_lab\doc\oop_in_c\encapsulation>
```

2. 继承

继承是基于现有类定义新类的能力，以便重用和组织代码。通过将继承的类属性结构作为派生类属性结构的第一个成员嵌入，可以轻松地在C中实现单个继承。

例如，不必从头创建shape类，您可以从现有的Shape类中继承大多数常见的内容，只添加与矩形不同的内容。下面是如何声明Rectangle类：

```
1  #ifndef RECT_H
2  #define RECT_H
3
4  #include "shape.h" /* the base class interface */
5
6  /* Rectangle's attributes... */
7  typedef struct {
8      Shape super; /* <== inherits Shape */
9
10     /* attributes added by this subclass... */
11     uint16_t width;
12     uint16_t height;
13 } Rectangle;
14
15 /* constructor prototype */
16 void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
17                     uint16_t width, uint16_t height);
18
19 #endif /* RECT_H */
```


Figure 2 Single inheritance in C: (a) class diagram with inheritance, and (b) memory layout for Rectangle and Shape objects

通过这种设计，您可以始终安全地将指向矩形的指针传递给任何需要指向形状的指针的C函数。具体来说，Shape类（称为超类或基类）中的所有函数都自动可用于矩形类（称为子类或派生类）。因此，不仅所有属性，而且来自超类的所有函数都被所有子类继承。

```

1  #include "rect.h"
2  // constructor implementation
3  void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
4                      uint16_t width, uint16_t height)
5  {
6      /* first call superclass' ctor */
7      Shape_ctor(&me->super, x, y);
8      /* next, you initialize the attributes added by this subclass... */
9      me->width = width;
10     me->height = height;
11 }
  
```

为了在C中严格正确，您应该在指向超类的指针上显式地强制转换指向子类的指针。在OOP中，这种转换称为向上转换，并且总是安全的。

```

1  #include "rect.h" /* Rectangle class interface */
2  #include <stdio.h> /* for printf() */
  
```

```
3
4 int main()
5 {
6     Rectangle r1, r2; /* multiple instances of Rect */
7
8     /* instantiate rectangles... */
9     Rectangle_ctor(&r1, 0, 2, 10, 15);
10    Rectangle_ctor(&r2, -1, 3, 5, 8);
11
12    printf("Rect r1(x=%d,y=%d,width=%d,height=%d)\n",
13          Shape_getX(&r1.super), Shape_getY(&r1.super),
14          r1.width, r1.height);
15    printf("Rect r2(x=%d,y=%d,width=%d,height=%d)\n",
16          Shape_getX(&r2.super), Shape_getY(&r2.super),
17          r2.width, r2.height);
18
19    /* re-use inherited function from the superclass Shape... */
20    Shape_moveBy((Shape *)&r1, -2, 3);
21    Shape_moveBy(&r2.super, 2, -1);
22
23    printf("Rect r1(x=%d,y=%d,width=%d,height=%d)\n",
24          Shape_getX(&r1.super), Shape_getY(&r1.super),
25          r1.width, r1.height);
26    printf("Rect r2(x=%d,y=%d,width=%d,height=%d)\n",
27          Shape_getX(&r2.super), Shape_getY(&r2.super),
28          r2.width, r2.height);
29
30    return 0;
31 }
```

正如你看到的，要调用继承的函数，您需要将第一个“me”参数显式向上强制转换到超类（Shape*），或者，您可以避免强制转换并使用成员“super”的地址（&r2->super）。

注意：对子类实例使用“继承的”函数没有额外的成本。换句话说，为子类的对象调用函数的成本与为超类的对象调用相同的函数的成本完全相同。这个开销也非常类似（实际上是一样的），如C++。

```

C:\>cd C:\qp_lab\doc\oop_in_c\inheritance

C:\qp_lab\doc\oop_in_c\inheritance>dir
Volume in drive C is Windows
Volume Serial Number is C007-D1C2

Directory of C:\qp_lab\doc\oop_in_c\inheritance

04/23/2018  11:50 AM    <DIR>          .
04/23/2018  11:50 AM    <DIR>          ..
04/23/2018  11:46 AM                2,055 main.c
02/03/2018  05:03 PM                 57 make.bat
04/23/2018  11:50 AM            49,403 oop_in_c.exe
02/03/2018  05:13 PM             1,412 rect.c
02/03/2018  05:13 PM             1,488 rect.h
04/23/2018  11:43 AM             1,527 shape.c
04/23/2018  11:41 AM             1,538 shape.h
              7 File(s)            57,480 bytes
              2 Dir(s)  301,953,630,208 bytes free

C:\qp_lab\doc\oop_in_c\inheritance>make

C:\qp_lab\doc\oop_in_c\inheritance>gcc shape.c rect.c main.c -o oop_in_c.exe

C:\qp_lab\doc\oop_in_c\inheritance>oop_in_c.exe
Rect r1(x=0,y=2,width=10,height=15)
Rect r2(x=-1,y=3,width=5,height=8)
Rect r1(x=-2,y=5,width=10,height=15)
Rect r2(x=1,y=2,width=5,height=8)

C:\qp_lab\doc\oop_in_c\inheritance>

```

3. 多态性（虚拟函数）

多态性是在运行时将匹配接口的对象相互替换的能力。C++实现了多态与虚拟函数。在C语言中，还可以通过多种方式实现虚拟函数^[1,4,10]。这里给出的实现（在QP/C和QP纳米实时框架中使用）与C++^[4,7,8]中的虚拟函数具有非常相似的性能和内存开销。

作为虚拟函数如何有用的一个例子，请再次考虑前面介绍的shape类。这个类可以提供更多有用的操作，例如area()（让形状计算自己的面积）或draw()（让形状在屏幕上绘制自己）等等。但问题是shape类不能提供这些操作的实现，因为shape太抽象，不“知道”如何计算，矩形子类（width * height）的计算结果与circle子类（pi * radius²）的计算结果非常不同。但是，这并不意味着Shape至少不能为操作提供接口，如Shape_area()或Shape_draw()，如下所示：

```

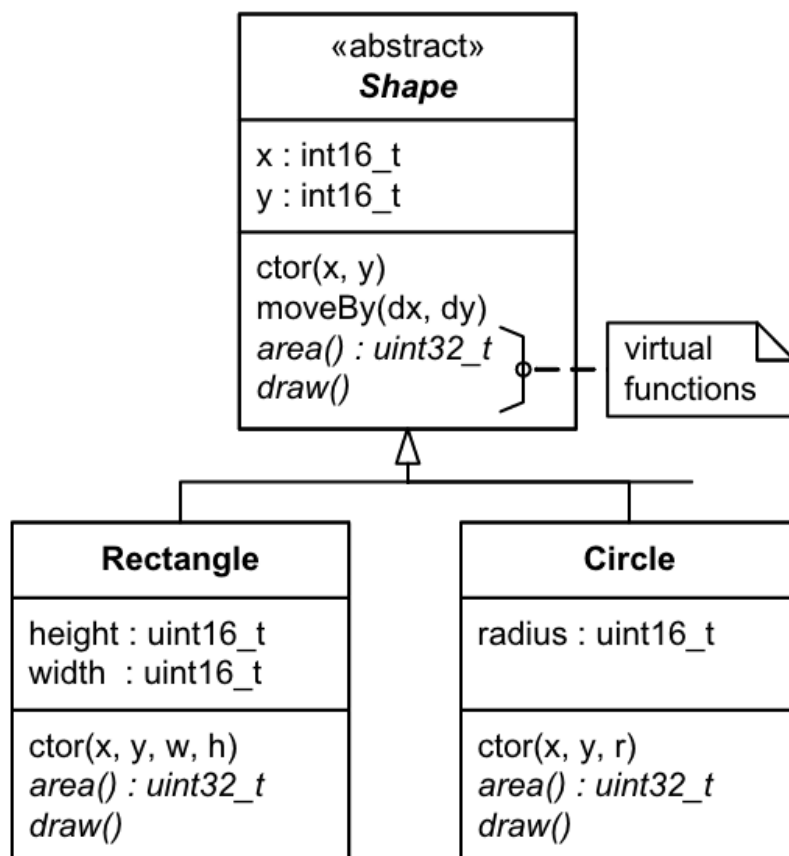
1  #ifndef SHAPE_H
2  #define SHAPE_H

```

```
3  #include <stdint.h>
4  /* Shape's attributes... */
5  struct ShapeVtbl; /* forward declaration */
6  typedef struct {
7      struct ShapeVtbl const *vptr; /* <== Shape's Virtual Pointer */
8      int16_t x; /* x-coordinate of Shape's position */
9      int16_t y; /* y-coordinate of Shape's position */
10 } Shape;
11
12 /* Shape's virtual table */
13 struct ShapeVtbl {
14     uint32_t (*area)(Shape const * const me);
15     void (*draw)(Shape const * const me);
16 };
17
18 /* Shape's operations (Shape's interface)... */
19 void Shape_ctor(Shape * const me, int16_t x, int16_t y);
20 void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);
21 int16_t Shape_getX(Shape const * const me);
22 int16_t Shape_getY(Shape const * const me);
23
24 static inline uint32_t Shape_area(Shape const * const me) {
25     return (*me->vptr->area)(me);
26 }
27
28 static inline void Shape_draw(Shape const * const me) {
29     (*me->vptr->draw)(me);
30 }
31
32 /* generic operations on collections of Shapes */
33 Shape const *largestShape(Shape const *shapes[], uint32_t nShapes);
34 void drawAllShapes(Shape const *shapes[], uint32_t nShapes);
35
36 #endif /* SHAPE_H */
```

事实上，这样的接口可能非常有用，因为它允许您编写通用代码来统一操作形状。例如，给定这样一个界面，您将能够编写一个通用函数来在屏幕上绘制所有形状或查找最大的形状（具有最大的面积）。在这一点上，这听起来可能有点理论化，但在本节后面的实际代码中，您将看到这一点。

Figure 3 Adding virtual functions `area()` and `draw()` to the `Shape` class and its subclasses



3.1 虚拟表 (vtbl) 和虚拟指针 (vptr)

现在应该很清楚，单个虚拟函数，例如 `Shape_area()`，在 `Shape` 的子类中可以有許多不同的实现。例如，`Shape` 的 `Rectangle` 子类的面积计算方法与 `Circle` 子类的不同。

这意味着虚拟函数调用无法在链接时解析，就像 C 中普通函数调用一样，因为要调用的函数的实际版本取决于对象的类型（矩形、圆形等），所以，相反，虚拟函数调用和实际实现之间的绑定必须在运行时发生，这称为后期绑定（与链接时绑定相反，链接时绑定也称为早期绑定）。

实际上，所有C++编译器都是通过每个类的一个虚拟表（VTBL）和每个对象的VirtualPosits（VPTR）来实现后期绑定[4， 7]。这种方法也适用于C语言。

虚表是与类引入的虚函数相对应的函数指针表。在C语言中，虚拟表可以由指向函数的指针结构模拟。

虚拟指针（vptr）是指向类的虚拟表的指针。该指针必须出现在类的每个实例（对象）中，因此它必须进入类的属性结构中。例如，Shape类的属性结构在顶部添加了vptr成员。

vptr被声明为指向不可变对象的指针（请参阅*前面的const关键字），因为虚拟表不应更改，实际上是在ROM中分配的。

虚拟指针（vptr）由所有子类继承，因此Shape类的vptr将自动在其所有子类中可用，如矩形、圆形等。

3.2 在构造函数中设置vptr

虚拟指针（vptr）必须初始化为指向类的每个实例（对象）中相应的虚拟表（vtbl）。执行此类初始化的理想位置是类的构造函数。事实上，这正是C++编译器生成VPTR隐式初始化的地方。

在C语言中，需要显式初始化vptr。以下是在形状的构造函数中设置vtbl和初始化vptr的示例：

```
1  #include "shape.h"
2  #include <assert.h>
3
4  /* Shape's prototypes of its virtual functions */
5  static uint32_t Shape_area_(Shape const * const me);
6  static void Shape_draw_(Shape const * const me);
7
8  /* constructor */
9  void Shape_ctor(Shape * const me, int16_t x, int16_t y) {
10     static struct ShapeVtbl const vtbl = { /* vtbl of the Shape class */
11         &Shape_area_,
12         &Shape_draw_
13     };
14     me->vptr = &vtbl; /* "hook" the vptr to the vtbl */
```

```
15     me->x = x;
16     me->y = y;
17 }
18
19 /* move-by operation */
20 void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy) {
21     me->x += dx;
22     me->y += dy;
23 }
24
25 /* "getter" operations implementation */
26 int16_t Shape_getX(Shape const * const me) {
27     return me->x;
28 }
29 int16_t Shape_getY(Shape const * const me) {
30     return me->y;
31 }
32
33 /* Shape class implementations of its virtual functions... */
34 static uint32_t Shape_area_(Shape const * const me) {
35     assert(0); /* purely-virtual function should never be called */
36     return 0U; /* to avoid compiler warnings */
37 }
38
39 static void Shape_draw_(Shape const * const me) {
40     assert(0); /* purely-virtual function should never be called */
41 }
42
43 /* the following code finds the largest-area shape in the collection */
44 Shape const *largestShape(Shape const *shapes[], uint32_t nShapes) {
45     Shape const *s = (Shape *)0;
46     uint32_t max = 0U;
47     uint32_t i;
48     for (i = 0U; i < nShapes; ++i) {
49         uint32_t area = Shape_area(shapes[i]); /* virtual call */
50         if (area > max) {
51             max = area;
52             s = shapes[i];
53         }
54     }
```

```

54     }
55     return s; /* the largest shape in the array shapes[] */
56 }
57
58 /* The following code will draw all Shapes on the screen */
59 void drawAllShapes(Shape const *shapes[], uint32_t nShapes) {
60     uint32_t i;
61     for (i = 0U; i < nShapes; ++i) {
62         Shape_draw(shapes[i]); /* virtual call */
63     }
64 }

```

如果一个类不能提供其某些虚拟函数的合理实现（因为这是一个抽象类，就像Shape一样），那么这些实现应该在内部断言。这样，您至少可以在运行时知道调用了一个未实现（纯虚拟）的函数。

3.3 继承vtbl并重写子类中的vptr

如前所述，如果一个超类包含vptr，那么它将由所有继承级别的所有派生子类自动继承，因此继承属性（通过“super”成员）的技术将自动适用于多态类。

但是，vptr通常需要重新分配给特定子类的vtbl。同样，这种重新赋值必须发生在子类的构造函数中。例如，这里是Shape的Rectangle子类的构造函数。

```

1  #include "rect.h" /* Rectangle class interface */
2  #include <stdio.h> /* for printf() */
3
4  /* Rectangle's prototypes of its virtual functions */
5  /* NOTE: the "me" pointer has the type of the superclass to fit the vtable */
6  static uint32_t Rectangle_area_(Shape const * const me);
7  static void Rectangle_draw_(Shape const * const me);
8
9  /* constructor */
10 void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
11                     uint16_t width, uint16_t height)
12 {
13     static struct ShapeVtbl const vtbl = { /* vtbl of the Rectangle class */

```



```

14     &Rectangle_area_,
15     &Rectangle_draw_
16 };
17 Shape_ctor(&me->super, x, y); /* call the superclass' ctor */
18 me->super.vptr = &vtbl; /* override the vptr */
19 me->width = width;
20 me->height = height;
21 }
22
23 /* Rectangle's class implementations of its virtual functions... */
24 static uint32_t Rectangle_area_(Shape const * const me) {
25     Rectangle const * const me_ = (Rectangle const *)me; /* explicit downcast */
26     return (uint32_t)me_>width * (uint32_t)me_>height;
27 }
28
29 static void Rectangle_draw_(Shape const * const me) {
30     Rectangle const * const me_ = (Rectangle const *)me; /* explicit downcast */
31     printf("Rectangle_draw_(x=%d,y=%d,width=%d,height=%d)\n",
32           Shape_getX(me), Shape_getY(me), me_>width, me_>height);
33 }

```

请注意，首先调用超类的构造函数（Shape_ctor（））来初始化从Shape继承的me->super。将vptr的构造函数设置为该形状的点。但是，vptr在下一个语句中被重写，在该语句中它被分配给矩形的vtb。

还请注意，虚拟函数的子类实现必须与超类中定义的签名精确匹配，以适合vtbl。例如，实现矩形区域（）采用类Shape * 的指针“me”，而不是它自己的类rectangle *。子类的实际实现必须执行“me”指针的显式向下转换。

3.4 虚拟调用（后期绑定）

有了虚拟表和虚拟指针的基础设施，虚拟调用（后期绑定）可以按如下方式实现：

```

1  uint32_t Shape_area(Shape const * const me)
2  {
3      return (*me->vptr->area)(me);
4  }

```

此函数定义可以放置在.c文件内，但缺点是每次虚拟调用都会产生额外的函数调用开销。为了避免这种开销，如果您的编译器支持内嵌函数（C99标准），您可以像这样将定义放入头文件中

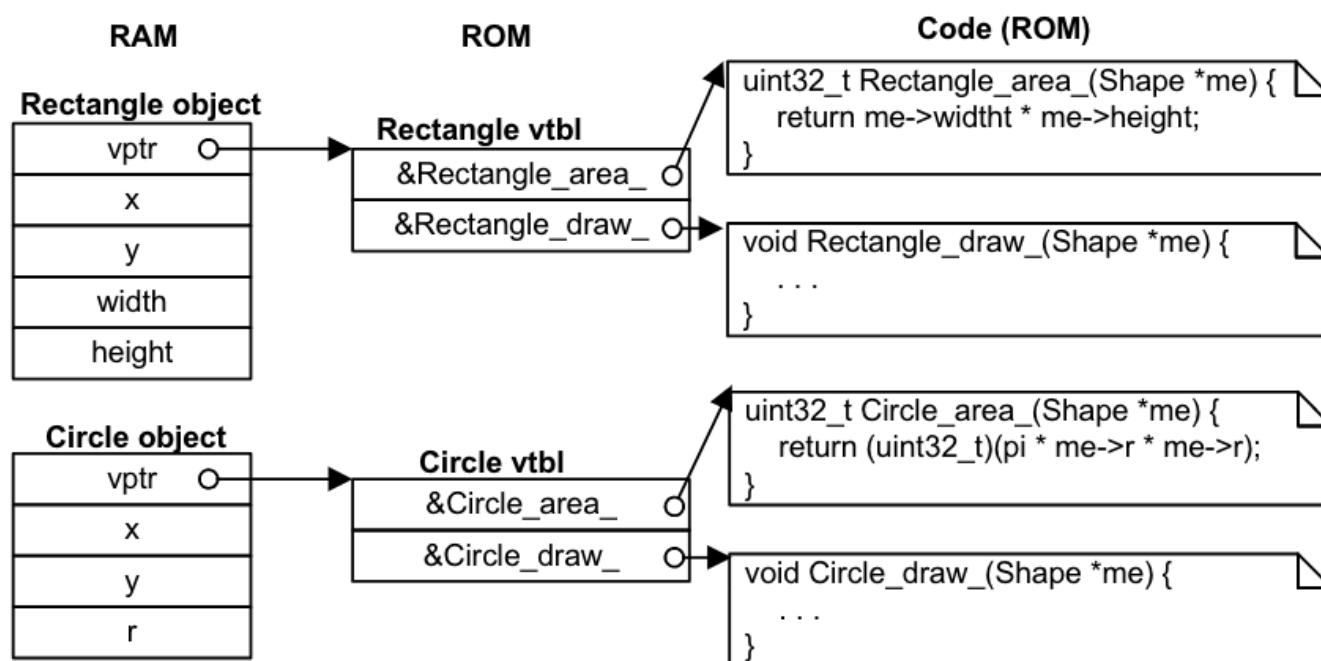
```
1 static inline uint32_t Shape_area(Shape const * const me)
2 {
3     return (*me->vptr->area)(me);
4 }
```

或者，对于较旧的编译器（C89），可以使用类似宏的函数，如下所示：

```
1 #define Shape_area(me_) ((*me_>vptr->area)((me_)))
```

无论哪种方式，虚拟调用的工作方式都是首先取消引用对象的vtbl以找到相应的vtbl，然后通过指向函数的指针从该vtbl调用适当的实现。下图说明了此过程：

Figure 4 Virtual Call Mechanism for Rectangles and Circles



3.5 使用虚函数的示例

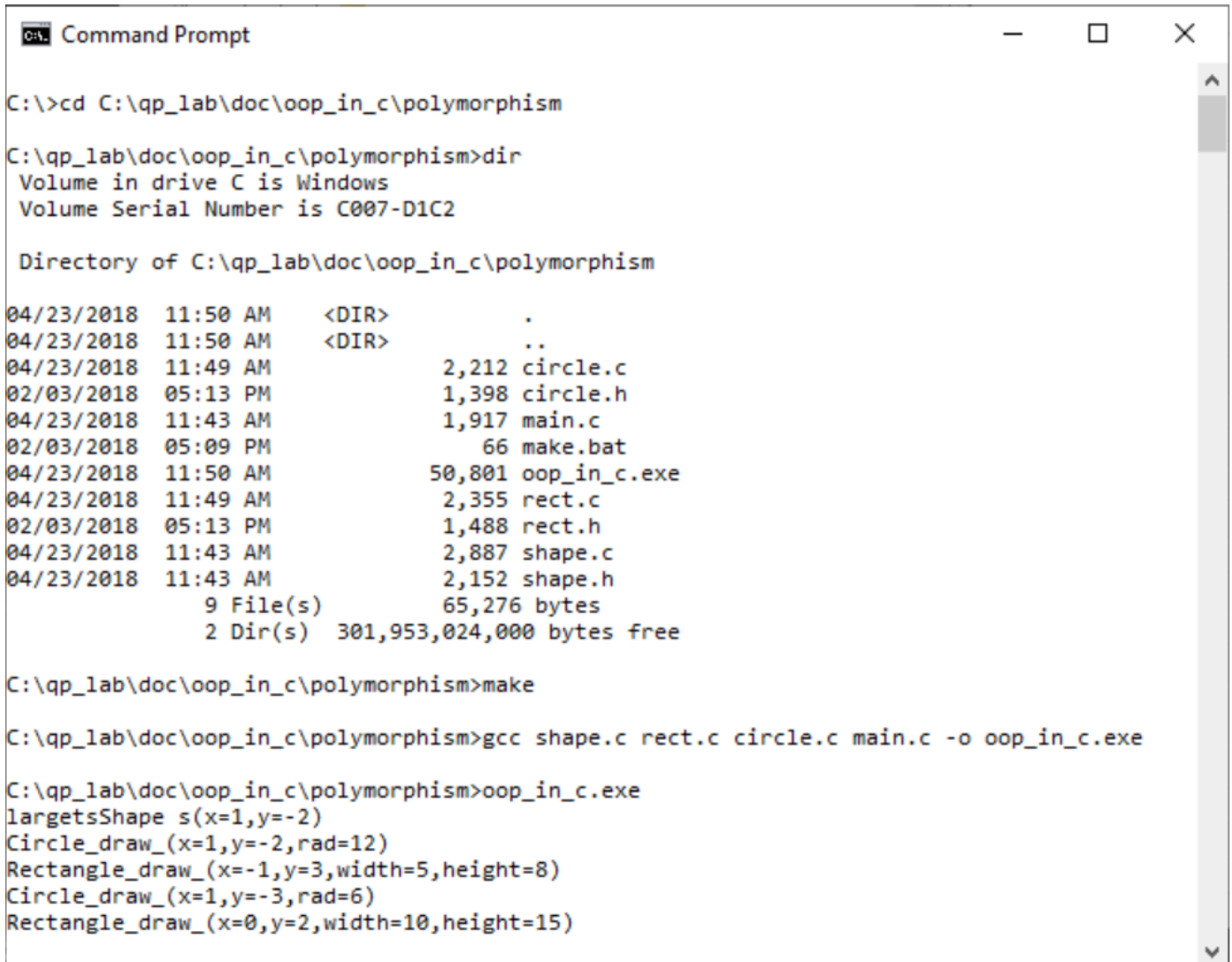
正如本节开头提到的多态性，虚拟函数允许您编写非常干净的泛型代码，并且独立于子类的特定实现细节。此外，代码自动支持开放的子类数量，这些子类可以在泛型代码开发（和编译）很久之后添加。

例如，shape中的两个函数，一个显示了在给定集合中查找最大面积形状的函数的通用实现，另一个显示了在给定集合中绘制所有形状的函数的通用实现。

下面的代码显示了如何使用所有这些功能。

```
1  #include "rect.h" /* Rectangle class interface */
2  #include "circle.h" /* Circle class interface */
3  #include <stdio.h> /* for printf() */
4
5  int main() {
6      Rectangle r1, r2; /* multiple instances of Rectangle */
7      Circle c1, c2; /* multiple instances of Circle */
8      Shape const *shapes[] = { /* collection of shapes */
9          &c1.super,
10         &r2.super,
11         &c2.super,
12         &r1.super
13     };
14     Shape const *s;
15
16     /* instantiate rectangles... */
17     Rectangle_ctor(&r1, 0, 2, 10, 15);
18     Rectangle_ctor(&r2, -1, 3, 5, 8);
19
20     /* instantiate circles... */
21     Circle_ctor(&c1, 1, -2, 12);
22     Circle_ctor(&c2, 1, -3, 6);
23
24     s = largestShape(shapes, sizeof(shapes)/sizeof(shapes[0]));
25     printf("largestShape s(x=%d,y=%d)\n",
26         Shape_getX(s), Shape_getY(s));
27
28     drawAllShapes(shapes, sizeof(shapes)/sizeof(shapes[0]));
```

```
29  
30     return 0;  
31 }
```



```
Command Prompt  
C:\>cd C:\qp_lab\doc\oop_in_c\polymorphism  
C:\qp_lab\doc\oop_in_c\polymorphism>dir  
Volume in drive C is Windows  
Volume Serial Number is C007-D1C2  
  
Directory of C:\qp_lab\doc\oop_in_c\polymorphism  
  
04/23/2018  11:50 AM    <DIR>          .  
04/23/2018  11:50 AM    <DIR>          ..  
04/23/2018  11:49 AM                2,212 circle.c  
02/03/2018  05:13 PM                1,398 circle.h  
04/23/2018  11:43 AM                1,917 main.c  
02/03/2018  05:09 PM                 66 make.bat  
04/23/2018  11:50 AM            50,801 oop_in_c.exe  
04/23/2018  11:49 AM                2,355 rect.c  
02/03/2018  05:13 PM                1,488 rect.h  
04/23/2018  11:43 AM                2,887 shape.c  
04/23/2018  11:43 AM                2,152 shape.h  
                9 File(s)              65,276 bytes  
                2 Dir(s)  301,953,024,000 bytes free  
  
C:\qp_lab\doc\oop_in_c\polymorphism>make  
C:\qp_lab\doc\oop_in_c\polymorphism>gcc shape.c rect.c circle.c main.c -o oop_in_c.exe  
C:\qp_lab\doc\oop_in_c\polymorphism>oop_in_c.exe  
targetShape s(x=1,y=-2)  
Circle_draw_(x=1,y=-2,rad=12)  
Rectangle_draw_(x=-1,y=3,width=5,height=8)  
Circle_draw_(x=1,y=-3,rad=6)  
Rectangle_draw_(x=0,y=2,width=10,height=15)
```

4. 总结

OOP是一种设计方法，而不是使用特定的语言或工具。本应用说明描述了如何在可移植的ANSI-C中实现封装、（单一）继承和多态性的概念。前两个概念（类和继承）的实现非常简单，没有增加任何额外的成本或开销。

多态性的结果是相当复杂的，如果你打算广泛使用它，你最好通过切换到C++。但是，如果您构建或使用库（如QP/C和QP nano实时框架），C中OOP的复杂性可能仅限于库，并且可以对应用程序开发人员有效隐藏。

5. 参考文献

[1]Miro Samek, “Portable Inheritance and Polymorphism in C”, Embedded Systems Programming December, 1997

[2]Miro Samek, “Practical Statecharts in C/C++”, CMP Books 2002, ISBN 978-1578201105

[3]Miro Samek, “Practical UML Statecharts in C/C++, 2nd Edition”, Newnes 2008, ISBN 978-0750687065

[4]Dan Saks, “Virtual Functions in C”, “Programming Pointers” column August, 2012, Embedded.com.

[5]Dan Saks, “Impure Thoughts”, “Programming Pointers” column September, 2012, Embedded.com.

[6]Dan Saks, “Implementing a derived class vtbl in C”, “Programming Pointers” column February, 2013, Embedded.com.

[7]Stanley Lippman, “Inside the C++ Object Model”, Addison Wesley 1996, ISBN 0-201-83454-5

[8]Bruce Eckel, “Thinking in C++”, <http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>

[9]StackOverflow: Object-Orientation in C, August 2011

[10]Axel-Tobias Schreiner, “Object-Oriented Programming in ANSI-C”, Hanser 1994, ISBN 3-446-17426-5