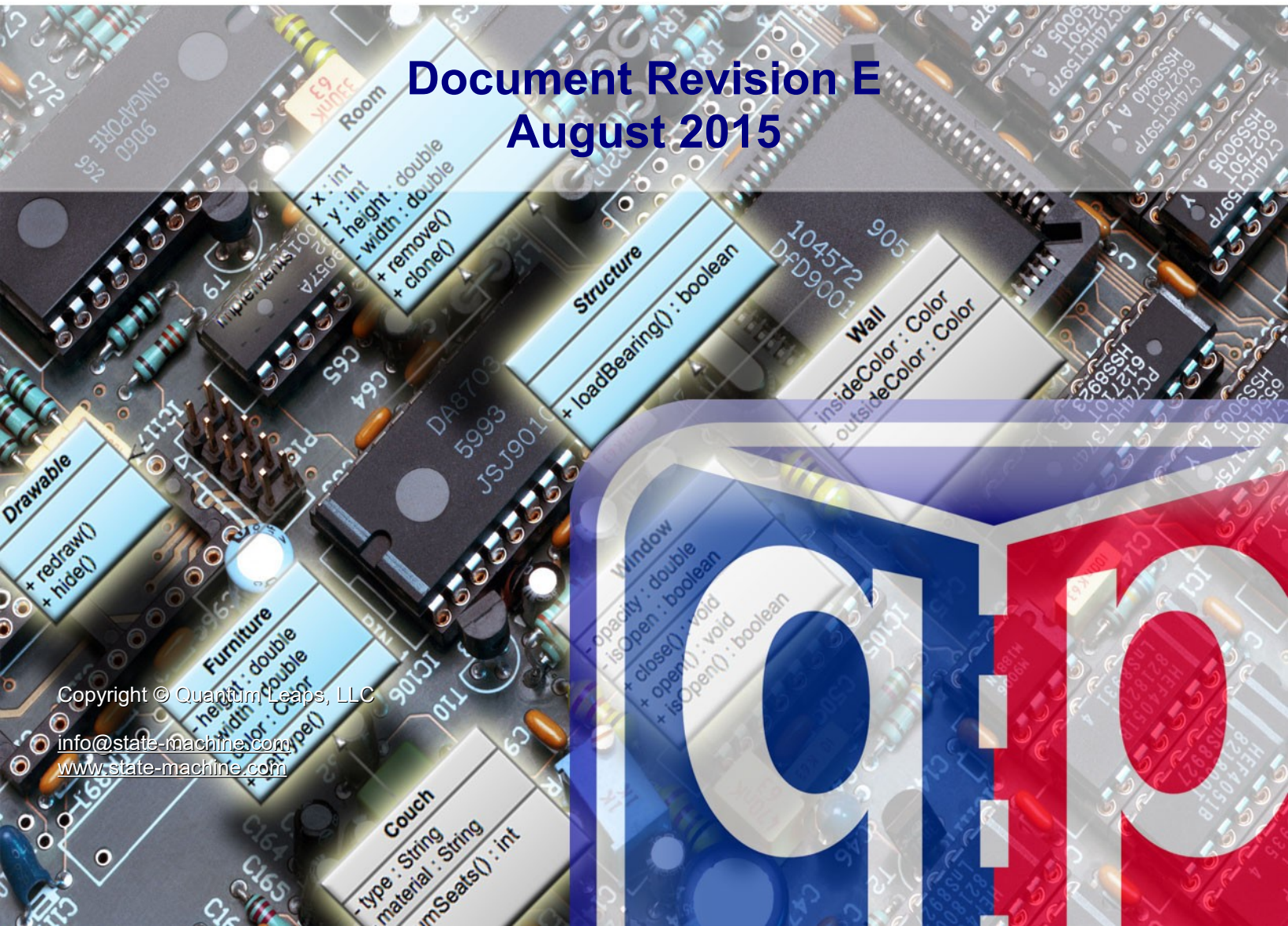
Quantum® LeaPs
innovating embedded systems

Application Note
Object-Oriented Programming in C

Document Revision E
August 2015



info@state-machine.com
www.state-machine.com

Table of Contents

1 Introduction

Object-oriented programming (OOP) is *not* the use of a particular language or a tool. It is rather a way of design based on the three fundamental design *meta*-patterns:

- **Encapsulation** – the ability to package data and functions together into *classes*
- **Inheritance** – the ability to define new classes based on existing classes in order to obtain reuse and code organization
- **Polymorphism** – the ability to substitute objects of matching interfaces for one another at run-time

Although these *meta*-patterns have been traditionally associated with object-oriented languages, such as Smalltalk, C++, or Java, you can implement them in almost any programming language including portable **ANSI-C** [1,2,3,4,5,6].

NOTES: If you simply develop end-user programs in C, but you also want to do OOP, you probably should be using C++ instead of C. Compared to C++, OOP in C can be cumbersome and error-prone, and rarely offers any performance advantage.

However, if you build software *libraries* or *frameworks* the OOP concepts can be very useful as the primary mechanisms of organizing the code. In that case, most difficulties of doing OOP in C can be confined to the library and can be effectively hidden from the application developers. This document has this primary use case in mind.

This Application Note describes how OOP is implemented in the QP/C and QP-nano active object (actor) frameworks. As a user of these frameworks, you need to understand the techniques, because you will need to apply them also to your own application-level code. But these techniques are not limited only to developing QP/C or QP-nano applications and are applicable generally to any C program.

---

2 Encapsulation

Encapsulation is the ability to package data with functions into **classes**. This concept should actually come as very familiar to any C programmer because it's quite often used even in the traditional C. For example, in the Standard C runtime library, the family of functions that includes `fopen()`, `fclose()`, `fread()`, and `fwrite()` operates on objects of type `FILE`. The `FILE` structure is thus *encapsulated* because client programmers have no need to access the internal attributes of the `FILE struct` and instead the whole interface to files consists only of the aforementioned functions. You can think of the `FILE` structure and the associated C-functions that operate on it as the `FILE` *class*. The following bullet items summarize how the C runtime library implements the `FILE` *class*:

1. Attributes of the class are defined with a C `struct` (the `FILE struct`).
2. Operations of the class are defined as C functions. Each function takes a pointer to the attribute structure (`FILE *`) as an argument. Class operations typically follow a common naming convention (e.g., all `FILE` class methods start with prefix `f`).
3. Special functions initialize and clean up the attribute structure (`fopen()` and `fclose()`). These functions play the roles of class constructor and destructor, respectively.

You can very easily apply these design principles to come up with your own "classes". For example, suppose you have an application that employs two-dimensional geometric shapes (perhaps to be rendered on an embedded graphic LCD). The basic `Shape` "class" in C can be declared as follows:

Listing 1 Declaration of the `Shape` "class" in C

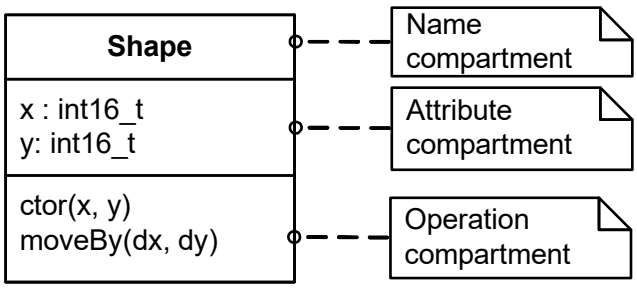
```
/* Shape's attributes... */
typedef struct {
    int16_t x; /* x-coordinate of Shape's position */
    int16_t y; /* y-coordinate of Shape's position */
} Shape;

/* Shape's operations (Shape's interface)... */
void Shape_ctor(Shape * const me, int16_t x, int16_t y);
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy);
```

The Shape "class" declaration goes typically into a header file (e.g., `shape.h`), although sometimes you might choose to put the declaration into a file scope (`.c` file).

One nice aspect of classes is that they can be drawn in diagrams, which show the class name, attributes, operations, and relationships among classes. The following figure shows the UML class diagram of the `Shape` class:

Figure 1 UML Class Diagram of the Shape class



And here is the definition of the `Shape`'s operations (must be in a `.c` file):

Listing 2 Definition of the `Shape` "class" in C

```
/* constructor */
void Shape_ctor(Shape * const me, int16_t x, int16_t y) {
    me->x = x;
    me->y = y;
}

/* move-by operation */
void Shape_moveBy(Shape * const me, int16_t dx, int16_t dy) {
    me->x += dx;
    me->y += dy;
}
```

You can create any number of `Shape` objects as instances of the `Shape` attributes `struct`. You need to initialize each instance with the "constructor" `Shape_ctor()`. You manipulate the `Shapes` only through the provided operations, which take the pointer "`me`" as the *first* argument.

NOTE: The "me" pointer in C corresponds directly to the implicit "this" pointer in C++. The "this" identifier is not used, because it is a keyword in C++ and such a program wouldn't compile with a C++ compiler.

Listing 3 Examples of using the `Shape` class in C

```
Shape s1, s2; /* multiple instances of Shape */

Shape_ctor(&s1, 0, 1);
Shape_ctor(&s2, -1, 2);
Shape_moveBy(&s1, 2, -4);
. . .
```

3 Inheritance

Inheritance is the ability to define new classes based on existing classes in order to reuse and organize code. You can easily implement single *inheritance* in C by literally embedding the inhertited class attribute structure as the *first* member of the derived class attribute structure.

For example, instead of creating a `Rectangle` class from scratch, you can inherit most what's common from the already existing `Shape` class and add only what's different for rectangles. Here's how you declare the `Rectangle` "class":

Listing 4 Declaration of the `Rectangle` as a Subclass of `Shape`

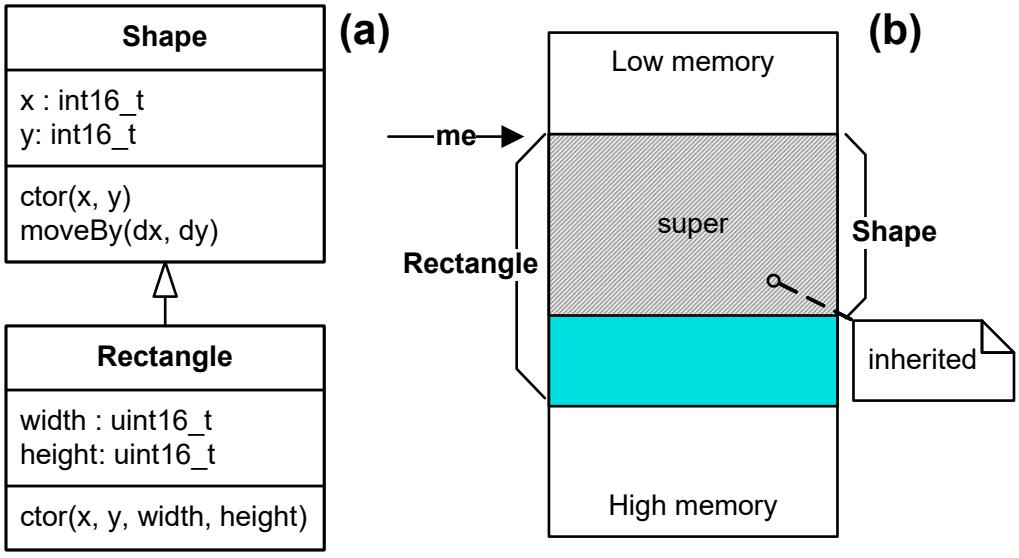
```
typedef struct {
    Shape super;     /* <== inherits Shape */
    /* attributes added by this subclass... */
    uint16_t width;
    uint16_t height;
} Rectangle;

/* constructor */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height);
```

As you can see, you implement inheritance by literally embedding the superclass (`Shape`) as the *first* member "`super`" of the subclass (`Rectangle`).

As shown in the following figure, this arrangement leads to the memory alignment, which lets you treat any pointer to the `Rectangle` class as a pointer to the `Shape` class:

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



NOTE: The alignment of the `Rectangle` structure and the inherited attributes from the `Shape` structure is guaranteed by the C Standard WG14/N1124. Section 6.7.2.1.13 of this Standard, says: *"… A pointer to a structure object, suitably converted, points to its initial member. There may be unnamed padding within a structure object, but not at its beginning"*.

With this arrangement, you can always safely pass a pointer to `Rectangle` to any C function that expects a pointer to `Shape`. Specifically, all functions from the `Shape` class (called the *superclass*) are automatically available to the `Rectangle` class (called the *subclass*). So, not only all attributes, but also all functions from the subclass are **inherited** by all subclasses.

Listing 5 The Constructor of class `Rectangle`

```
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height)

{
    /* first call superclass' ctor */
    Shape_ctor(&me->super, x, y);

    /* next, you initialize the attributes added by this sub-class... */
    me->width  = width;
    me->height = height;
}
```

To be strictly correct in C, you should explicitly cast a pointer to the subclass on the pointer to the superclass. In OOP such casting is called *upcasting* and is always safe.

Listing 6 Example of Using `Rectangle` Objects

```
Rectangle r1, r2;

/* instantiate rectangles... */
Rectangle_ctor(&r1, 0, 2, 10, 15);
Rectangle_ctor(&r2, -1, 3, 5, 8);

/* re-use inherited function from superclass Shape... */
Shape_moveBy((Shape *)&r1, -2, 3);
Shape_moveBy(&r2->super, 2, -1);
```

As you can see, to call the inherited functions you need to either explicitly up-cast the first "`me`" parameter to the superclass `(Shape *)`, or alternatively, you can avoid casting and take the address of the member "`super`" (`&r2->super`).

NOTE: There are no additional costs to using the "inherited" functions for instances of the subclasses. In other words, the overhead of calling a function for an object of a subclass is exactly as expensive as calling the same function for an object of the superclass. This overhead is also very similar (identical really) as in C++.

4 Polymorphism (Virtual Functions)

Polymorphism is the ability to substitute objects of matching interfaces for one another at run-time. C++ implements polymorphism with *virtual* functions. In C, you can also implement virtual functions in a number of ways [1,4,10]. The implementation presented here (and used in the QP/C and QP-nano active object frameworks) has very similar performance and memory overhead as virtual functions in C++ [4,7,8].

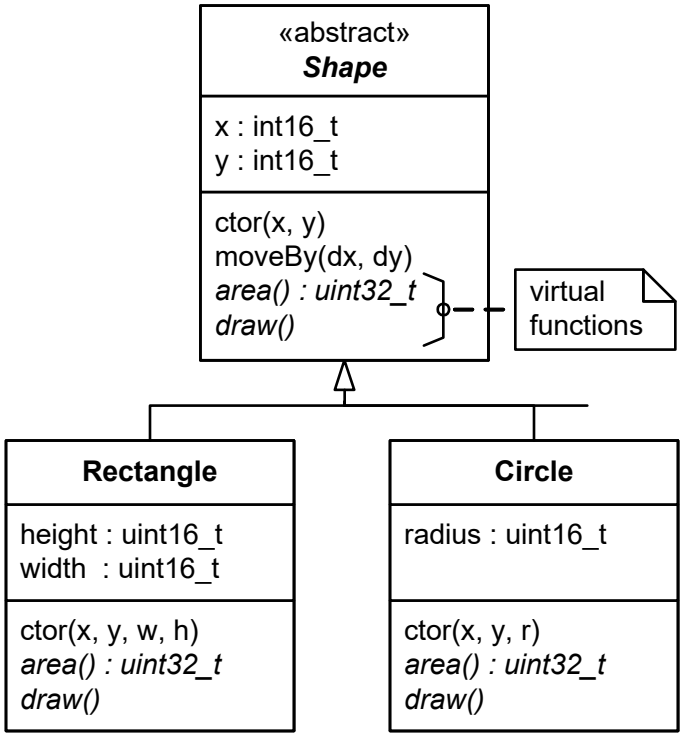
As an example of how virtual functions could be useful, consider again the `Shape` class introduced before. This class could provide many more useful operations, such as `area()` (to let the shape compute its area) or `draw()` (to let the shape draw itself on the screen), etc. But the trouble is that the `Shape` class cannot provide the actual implementation of such operations, because `Shape` is too *abstract* and doesn't "know" how to calculate, say its own area. The computation will be very different for a `Rectangle` subclass (width * height) than for the `Circle` subclass (pi * radius^2).

However, this does not mean that `Shape` cannot provide at least the *interface* for the operations, like `Shape_area()` or `Shape_draw()`, as follows:

```
uint32_t Shape_area(Shape * const me);
void Shape_draw(Shape * const me);
```

In fact, such an interface would be very useful, because it would allow you to write generic code to manipulate shapes uniformly. For example, given such an interface, you will be able to write a generic function to draw all shapes on the screen or to find the largest shape (with the largest area). This might sound a bit theoretical at this point, but it will become more clear when you see the actual code later in this Section.

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



4.1 Virtual Table (`vtbl`) and Virtual Pointer (`vptr`)

By now it should be clear that a single virtual function, such as `Shape_area()`, can have many different implementations in the subclasses of `Shape`. For example, the `Rectangle` subclass of `Shape` will have a different way of calculating its area than the `Circle` subclass.

This means that a virtual function call cannot be resolved at link-time, as it is done for ordinary function calls in C, because the actual version of the function to call depends on the *type* of the object (`Rectangle`, `Circle`, etc.) So, instead the binding between the invocation of a virtual function and the actual implementation must happen at run-time, which is called *late binding* (as opposed to the link-time binding, which is also called *early binding*).

Practically all C++ compilers implement late binding by means of one Virtual Table (`vtbl`) per class and a Virtual Pointer (`vptr`) per each object [4,7]. This method can be applied to C as well.

Virtual Table is a table of function pointers corresponding to the virtual functions introduced by the class. In C, a Virtual Table can be emulated by a structure of pointers-to-function, as follows:

Listing 7 Virtual Table for the `Shape` Class (see also Figure 3)

```
struct ShapeVtbl {
    uint32_t (*area)(Shape * const me);
    void (*draw)(Shape * const me);
};
```

Virtual Pointer (`vptr`) is a pointer to the Virtual Table of the class. This pointer must be present in every instance (object) of the class, and so it must go into the attribute structure of the class. For example, here is the attribute structure of the `Shape` class augmented with the `vptr` member added at the top:

Listing 8 Adding the Virtual Pointer (`vptr`) to the `Shape` class

```
/* Shape's attributes... */
struct ShapeVtbl; /* forward declaration */
typedef struct {
    struct ShapeVtbl const *vptr; /* <== Shape's Virtual Pointer */
    int16_t x; /* x-coordinate of Shape's position */
    int16_t y; /* y-coordinate of Shape's position */
} Shape;
```

The vptr is declared as pointer to an immutable object (see the `const` keyword in front of the *), because the Virtual Table should not be changed and is, in fact, allocated in ROM.

The Virtual Pointer (`vptr`) is inherited by all subclasses, so the vptr of the `Shape` class will be automatically available in all its subclasses, such as `Rectangle`, `Circle`, etc.

4.2 Setting the `vptr` in the Constructor

The Virtual Pointer (`vptr`) must be initialized to point to the corresponding Virtual Table (`vtbl`) in every instance (object) of a class. The ideal place to perform such initialization is the class' constructor. In fact, this is exactly where the C++ compilers generate an implicit initialization of the `vptr`.

In C, you need to initialize the `vptr` explicitly. Here is an example of setting up the `vtbl` and the initialization of the `vptr` in the `Shape`'s constructor:

Listing 9 Defining the Virtual Table and Initializing the Virtual Pointer (`vptr`) in the constructor

```
/* Shape class implementations of its virtual functions... */
static uint32_t Shape_area_(Shape * const me);
static void Shape_draw_(Shape * const me);

/* constructor */
void Shape_ctor(Shape * const me, int16_t x, int16_t y) {
    static struct ShapeVtbl const vtbl = { /* vtbl of the Shape class */
        &Shape_area_,
        &Shape_draw_
    };
    me->vptr = &vtbl; /* "hook" the vptr to the vtbl */

    me->x = x;
    me->y = y;
}
```

As you can see the `vtbl` is defined as both `static` and `const`, because you need only one instance of `vtbl` per class and also the `vtbl` should go into ROM (in embedded systems).

The `vtbl` is initialized with pointer to functions that implement the corresponding operations. In this case, the implementations are `Shape_area_()` and `Shape_draw_()` (please note the trailing underscores).

If a class cannot provide a reasonable implementation of some of its virtual functions (because this is an abstract class, as `Shape` is), the implementations should assert internally. This way, you would know at least at run-time, that an unimplemented (purely virtual) function has been called:

Listing 10 Defining purely virtual functions for the `Shape` class

```
/* Shape class implementations of its virtual functions... */
static uint32_t Shape_area_(Shape * const me) {
    ASSERT(0); /* purely-virtual function should never be called */
    return 0U; /* to avoid compiler warnings */
}

static void Shape_draw_(Shape * const me) {
    ASSERT(0); /* purely-virtual function should never be called */
}
```

4.3 Inheriting the `vtbl` and Overriding the `vptr` in the Subclasses

As mentioned before, if a superclass contains the `vptr`, it is inherited automatically by all the derived subclasses at all levels of inheritance, so the technique of inheriting attributes (via the "`super`" member) works automatically for polymorphic classes.

However, the `vptr` typically needs to be re-assigned to the `vtbl` of the specific subclass. Again, this re-assignment must happen in the subclass' constructor. For example, here is the constructor of the `Rectangle` subclass of `Shape`:

Listing 11 Overriding the `vtbl` and `vptr` in the subclass `Rectangle` of the `Shape` superclass

```
/* Rectangle's class implementations of its virtual functions... */
static uint32_t Rectangle_area_(Shape * const me);
static void Rectangle_draw_(Shape * const me);

/* constructor */
void Rectangle_ctor(Rectangle * const me, int16_t x, int16_t y,
                    uint16_t width, uint16_t height)
{
    static struct ShapeVtbl const vtbl = { /* vtbl of the Rectangle class */
        &Rectangle_area_,
        &Rectangle_draw_
    };
    Shape_ctor(&me->super, x, y); /* call the superclass' ctor */
    me->super.vptr = &vtbl; /* override the vptr */

    me->width  = width;
    me->height = height;
}
```

Please note that the superclass' constructor (`Shape_ctor()`) is called first to initialize the me->super member inherited from `Shape`. This constructor sets the `vptr` to point to the `Shape`'s `vtbl`. However, the `vptr` is overridden in the next statement, where it is assigned to the `Rectangle`'s `vtbl`.

Please also note the the subclass' implementations of the virtual functions must match the signatures defined in the superclass exactly in order to fit into the `vtbl`. For example, the implementation `Rectangle_area_()` takes the pointer "`me`" of class `Shape*`, instead of its own class `Rectangle*`. The actual implementation from the subclass must then perform an explicit downcast of the "`me`" pointer, as illustrated below:

Listing 12 Explicit downcasting of the "`me`" pointer in the subclass' implementation

```
static uint32_t Rectangle_area_(Shape * const me) {
    Rectangle * const me_ = (Rectangle *)me; /* explicit downcast */
    return (uint32_t)me_->width * (uint32_t)me_->height;
}
```

NOTE: To simplify the discussion, Listing 11 shows the case where `Rectangle` does not introduce any new virtual functions of its own. In this case, `Rectangle` can just re-use the `ShapeVtbl` "as is". However, it is also fairly straightforward to extend the implementation to the generic case where `Rectangle` would introduce its own `RectangleVtbl` that would inherit `ShapeVtbl`.

4.4 Virtual Call (Late Binding)

With the infrastructure of Virtual Tables and Virtual Pointers in place, the virtual call (late binding) can be realized as follows:

```
uint32_t Shape_area(Shape * const me) {
    return (*me->vptr->area)(me);
}
```

This function definition can be either placed in the `.c` file scope, but the downside is that you incur additional function call overhead for every virtual call. To avoid this overhead, if your compiler supports in-lining of functions (C99 standard), you can put the definition in the header file like this:

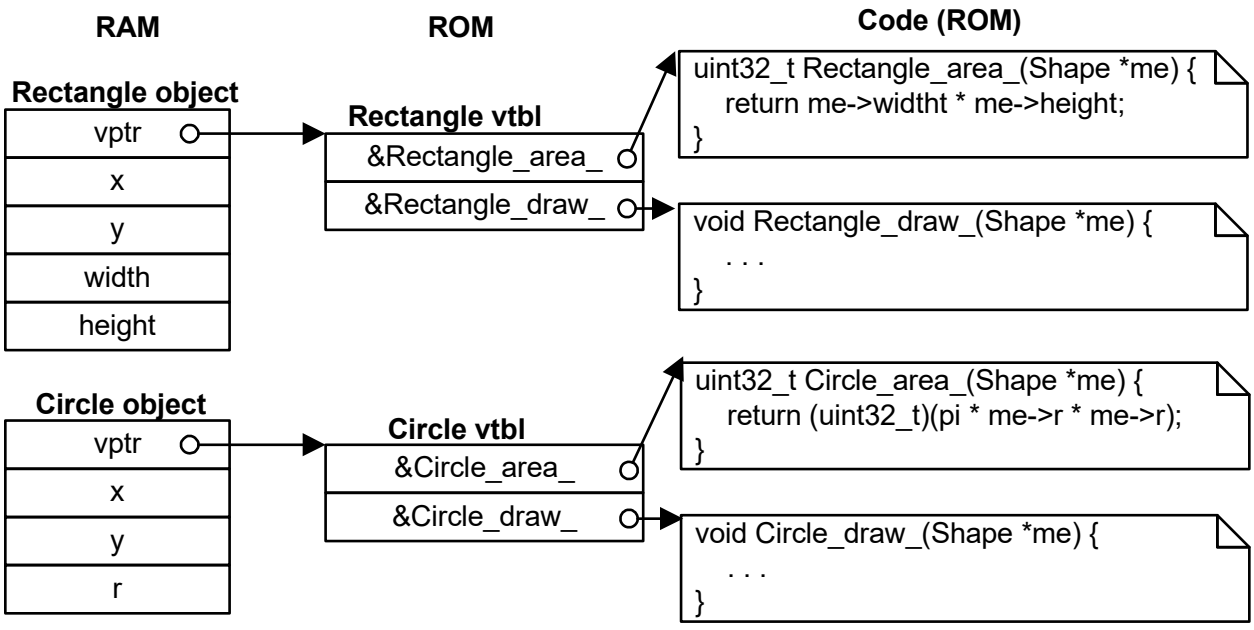
```
static inline uint32_t Shape_area(Shape * const me) {
    return (*me->vptr->area)(me);
}
```

Alternatively, for older compilers (C89) you can use function-like macro, like this:

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

Either way, the virtual call works by first de-referencing the `vtbl` of the object to find the corresponding `vtbl`, and only then calling the appropriate implementation from this `vtbl` via a pointer-to-function. The figure below illustrates this process:

Figure 4 Virtual Call Mechanism for Rectangles and Circles



4.5 Examples of Using Virtual Functions

As mentioned in the beginning of the section on polymorphism, virtual functions allow you to write generic code that is very clean and independent on the specific implementation details for subclasses. Moreover, the code automatically supports an open-ended number of sub-classes, which can be added long after the generic code has been developed.

For example, the following code finds and returns the largest shape on the screen:

```
Shape const *largestShape(Shape const *shapes[], size_t nShapes) {
    Shape const *s = NULL;
    uint32_t max = 0U;
    size_t i;
    for (i = 0U; i < nShapes; ++i) {
        uint32_t area = Shape_area(shapes[i]); /* virtual call */
        if (area > max) {
            max = area;
            s = shape[i];
        }
    }
    return s; /* the largest shape in the array shapes[] */
}
```

Similarly, the following code will draw all `Shape`s on the screen:

```
void drawAllShapes(Shape const *shapes[], size_t nShapes) {
    size_t i;
    for (i = 0U; i < nShapes; ++i) {
        Shape_draw(shapes[i]); /* virtual call */
    }
}
```

5 Summary

OOP is a design method rather than the use of a particular language or a tool. This Application Note described how to implement the concepts of encapsulation, (single) inheritance, and polymorphism in portable ANSI-C. The first two of these concepts (classes and inheritance) turned out to be quite simple to implement without adding any extra costs or overheads.

Polymorphism turned out to be quite involved, and if you intend to use it extensively, you would be probably better off by switching to C++. However, if you build or use application frameworks, such as the the QP/C and QP-nano active object (actor) frameworks, the complexities of the OOP in C can be confined to the framework and can be effectively hidden from the application developers.

6 References

[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

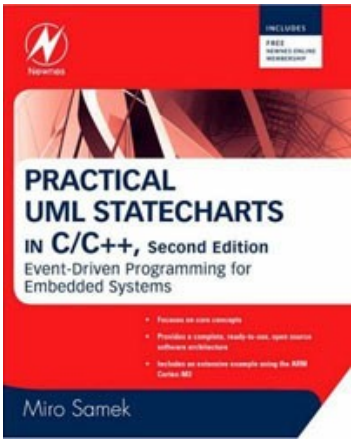
[10] Axel-Tobias Schreiner, “Object-Oriented Programming in ANSI-C”, Hanser 1994, ISBN 3-446-17426-5

7 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 919 360-5668
+1 919 869-2998 (FAX)

Email: info@state-machine.com
Web : http://www.state-machine.com/



"*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*", by Miro Samek, Newnes, 2008