by MIRO SAMEK

Portable Inheritance and Polymorphism in C

Although object-oriented designs are largely language-independent, most literature assumes C++, Smalltalk, or Java for OO implementations. Here's a lower-level view that assumes only a procedural language like C for embedded developers who want to apply OO without switching to an OO language.

Is it possible to write object-oriented (OO) programs in a non-OO language like C? How can you implement an OO design in a small embedded system without a C++ compiler available? How can you improve your coding style in C to make your code more re-usable, modular, and robust? How do inheritance and polymorphism actually work? Is the overhead acceptable in your system? How much convenience and expressiveness do you have to compromise by implementing OO design in C rather than in an OO language? In this article I address these questions by presenting a small, portable, and efficient C language implementation of the following OO concepts:

- Encapsulation—packaging data with functions into classes, as well as techniques for information hiding and modularity
- Inheritance—the ability to define new classes and behavior based on existing classes to obtain code re-use and code organization
- Polymorphism—the same message sent to different objects, which results in behavior that is dependent on the nature of the object receiving the message

In my implementation I adopt a Java language approach to inheritance and polymorphism.[1] Class inheritance (or implementation inheritance) is provided only as a single inheritance model with the object abstract class at the root of the class hierarchy. In contrast, the implementation permits multiple implementation inheritance, allowing classes to implement many Java-style interfaces.

In spite of unquestioned advantages of OO languages, C still remains the

In spite of the unquestioned advantages of OO languages, C still remains the best known implementation language.

best known and widespread implementation language. Many embedded systems today still simply do not offer any other choice. Consequently, most developers still exclusively use procedural programming techniques, and many are unaware that it's quite easy to implement key OO concepts directly in C. Promoting this awareness may be important for at least two reasons.

The first is leveraging OO technology. Most OO designs can be implemented in C, but many developers wouldn't consider them without the availability of an OO language. This unnecessarily limits application of the technology.

The second is smoothing transition from procedural to OO thinking. Migrating to OO technology can require quite a leap in your way of thinking. Implementing OO concepts in currently used and familiar languages gives you an opportunity to get exposed to the new programming paradigm right away and without major investment.

ENCAPSULATION

You can achieve packaging of data with functions in C by making each class attribute (instance variable) a field in the C `struct`. You implement class methods as C functions that take as their first argument the pointer to the structure of attributes (the `this` pointer). You can further strengthen the association between the attributes and methods by a consistent naming convention for method names. The most popular convention that I adopt is to concatenate the structure name (class name) with the operation name. This altering of function names is part of *name decorating* (also known as name mangling) and is performed implicitly by most C++ compilers. Because name decorating eliminates method name clashes between different classes, it effectively partitions the flat C function namespace into separate namespaces nested within classes.

The next aspect I address by a coding convention is access control. In C, you can only indicate your intention for the level of access permitted to a particular attribute or a method. Conveying this intention through the name of an attribute or a method is better than just expressing it in the form of a comment at the declaration point. In this way, unintentional access to class members in any portion of the code will be easier to detect. Most OO designs distinguish the following three levels of protection:

- private—accessible only from within the class
- protected—accessible only by the class and its subclasses
- public—available anywhere (default in C)

I use a double underscore prefix (`__foo`) for private attributes. Note that there is usually no need to expose private methods (helper methods) in the class declaration file (.H file). Rather, you should hide them completely in the implementation file (declare them static in the .C file). For protected members I use a single underscore (`_foo`, `String_Foo`). I avoid using underscores in the public members at all (`foo`, `StringFoo`). So, with this convention, the presence of underscores in a name is a signal that access rights should be checked against the context of use. Because public members can be used without any constraints, they don't need any special adornment.

Every class must provide at least one constructor method for initialization of its attribute structure. Constructor calls should be the only method of initialization. Otherwise, the internal structure of an object must be

exposed, thereby compromising encapsulation.

Optionally, a class may provide a destructor, which is a method responsible for releasing the resources allocated during the lifetime of an object. Whereas there may be many ways of instantating a class (different constructors taking different arguments), there should be only one way of destroying an object.

Because of the special role of constructors and destructors, I again advise a consistent naming pattern. I use base names "Con" (`FooCon1`, `FooCon2`) and "Des" (`FooDes`) for constructors and destructors, respectively. I suggest that constructors return either the pointer to a properly initialized attribute structure, or `NULL` if initialization fails. The destructor should take only the `this` argument and should return `void`.

Objects can be allocated statically, dynamically (on the heap), or automatically (on the stack). Because of C syntax limitations, you generally cannot initialize objects through a constructor call at the definition point. For static objects, you cannot call a constructor at all, because function calls aren't permitted in a static initializer. Automatic objects must all be defined at the beginning of a block and at this point, you will generally not have enough initialization information to call an appropriate constructor. Therefore, you will often have to divorce object allocation from initialization. You should treat objects as all other C variables, in that you never use them before initialization. Typically, you'll want to initialize objects as soon as the initialization information becomes available.

Some objects may require destruction, and calling destructors for all objects when they become obsolete or go out of scope is a good programming practice. Later, I'll show that a virtual destructor is available for all classes.

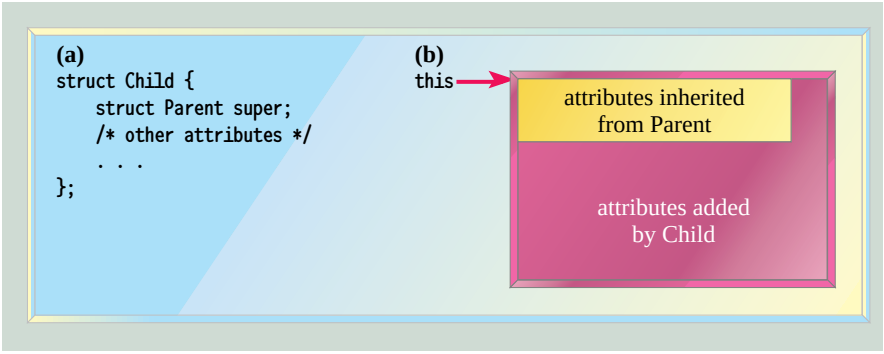
INHERITANCE

Inheritance is a mechanism by which new and more specialized classes can be defined in terms of existing classes. When a child class (subclass) inherits from a parent class (superclass), the subclass then includes the definitions of all the attributes and methods that the superclass defines. Usually, the subclass extends the superclass by adding its own attributes and methods. Objects that are instances of the subclass contain all data defined by the subclass *and* its parent classes, and they are able to perform all operations defined by this subclass *and* its parents.

This kind of class relationship can be implemented in C by embedding the parent class attribute structure as the

Portable Inheritance and Polymorphism

FIGURE 1
Declaration of "Child" attribute structure with embedded "Parent" as the first member "super" (a); Memory layout (b).



first member of the child class structure. As shown in Figure 1, this structuring results in such an attribute alignment that a pointer to the Child class can always be safely cast (upcast) on the pointer to the Parent class. In particular, this pointer can always be passed to any C function expecting a pointer to the Parent class. (To be strictly correct in C, you should explicitly upcast this pointer.) This means that all methods of parent classes are automatically available to child classes—in other words, they are inherited.

This simple approach works only for single inheritance (one parent class) because a class with many parent classes cannot align attributes with all of those parents.

I've named the inherited member `super` to make the inheritance relationship between classes more explicit and to increase the similarity to Java. The `super` member provides a handle to access the superclass attributes. For example, a grandchild class can access its grandparent protected attribute `_foo` as `this-> super.super._foo`.

Inheritance adds responsibilities to class constructors and the destructor. Because each child object contains an embedded parent object, the child constructor must take care of initializing the portion controlled by the parent. To avoid any potential dependencies, the superclass constructor should be called before initializing the attributes. Exactly the opposite holds true for the destructor. The inherited portion should be destroyed as the last step.

In my implementation, I also adopt from Java the concept of a single abstract base class `Object`. This means that no class can be defined as standalone, but rather must extend some other class, with the `Object` class at the root of the class hierarchy. This setup is especially convenient in this hybrid implementation (OO add-on to a procedural language) because every object can be ultimately treated as the `Object` class instance—which clearly separates objects from all other types. This differs from the C++ approach, in which each structure is equivalent to a class. As I will demonstrate, my `Object` class adds important behavior, subsequently inherited by all other classes, thus enabling polymorphism.

LISTING 1
Declaration of `String` *class.*

```
#include "object.h"
/** Character String class */
CLASS(String, Object)
   char *__buffer; /* private buffer */
VTABLE(String, Object)
METHODS
   /* public constructors */
   String StringCon1(String this,
                     const char *str);
   String StringCon2(String this,
                     String other);

   /* public destructor */
   void StringDes(String this);

   /* public to char conversion */
   const char *StringToChar(String this);
END_CLASS
```

Listing 1 shows the declaration of a basic class `String` extending the class `Object`. The class encapsulates a character buffer (`__buffer`), provides two ways of construction (`StringCon1`, `StringCon2`), a destructor, and a method for read-only access to the character buffer `StringToChar`. The class is declared by means of preprocessor macros: `CLASS`, `VTABLE`, `METHODS`, and `END_CLASS`, which are defined and documented in object.h, available to download at *www.embedded.com/code*.

POLYMORPHISM

An extended class often overrides the behavior of its superclass by providing new implementations of one or more inherited methods. For example, class `Object` defines destructor `Object_Des`. Class `String`, which extends `Object` (refer to Listing 1), overrides this behavior with its own destructor, `StringDes`. Let's assume that somewhere in your code you destroy a heterogeneous container holding generic `Object` pointers. Because `String` (as all other classes) inherits from `Object`, some pointers from the collection may actually point to `String` objects. Your code will be polymorphic if it invokes the correct implementation `StringDes` to destroy `String` objects (and possibly other implementations for objects of other classes). Polymorphic behavior requires method resolution depending on the run-time class of the object (`String`) and not the class of the pointer (`Object`), and is known as dynamic binding.

You can efficiently implement dynamic binding in C by introducing an additional level of indirection in method resolution. Rather than calling a method (C function) directly, you call a function pointed to by a function pointer defined in a *class descriptor* referenced by each object.[2] The class

descriptor (sometimes called the virtual table or VTABLE)[3] is a record of function pointers corresponding to virtual functions—in other words, methods intended to be overridden by subclasses.

In the previous example, the `Object` class implementation of a virtual destructor looks as follows. The class descriptor of the `Object` class declares function pointer `Des`:

```
struct ObjectClass {
  ...
 void (*Des)(struct Object*);
};
```

Each instance of class `Object` maintains a pointer (called a virtual pointer or VPTR—see Eckel, 1995) to this class descriptor:

```
struct Object {
  struct ObjectClass *__vptr;
};
```

Dynamic binding for the virtual destructor now takes the form:

```
(*obj->__vptr->Des)(obj);
```

where `obj` points to the `Object` structure.

Note that the pointer `obj` is used here twice: once for resolving the method and once as the `this` argument. Dynamic binding requires two more memory accesses and one more addition than a direct function call (Rumbaugh, 1991). The memory cost of late binding is storing the virtual pointer (inherited from `Object`) in each object, plus the cost of storing one VTABLE per class.

Class descriptors can themselves be treated as sole instances of VTABLE classes (a class being represented as a VTABLE object). Therefore you can apply the technique of nesting VTABLEs to achieve inheritance of virtual functions. This is encapsulated in the macro `VTABLE` (see Listing 1). All class descriptors inherit directly or indirectly from the `ObjectClass` descriptor, so all inherit the virtual destructor.

Inheritance slightly complicates the syntax of virtual function calls. In general, you will have to upcast the object pointer (on the `Object` class) and downcast the virtual pointer `__vptr` (on the specific class descriptor). These operations, as well as double object pointer referencing, are encapsulated in the macros `VCALL` and `END_CALL`. For example, the virtual destructor call for object `obj` of *any* class takes the form:

```
VCALL(obj, Object, Des)END_CALL;
```

If a virtual function takes arguments other than `this`, they should be listed before macro `END_CALL`. For example:

```
result = VCALL(obj, FooClass, Foo) ,5 ,i+j
         END_CALL;
```

where `obj` points to a `FooClass` or any subclass of `FooClass` and virtual function `Foo` is defined in `FooClass` VTABLE.

Virtual tables need to be initialized through their constructors. The initialization performed by the VTABLE constructor can be broken into two steps: copying the inherited VTABLE, and customizing it by overriding the implementation of the chosen virtual functions.

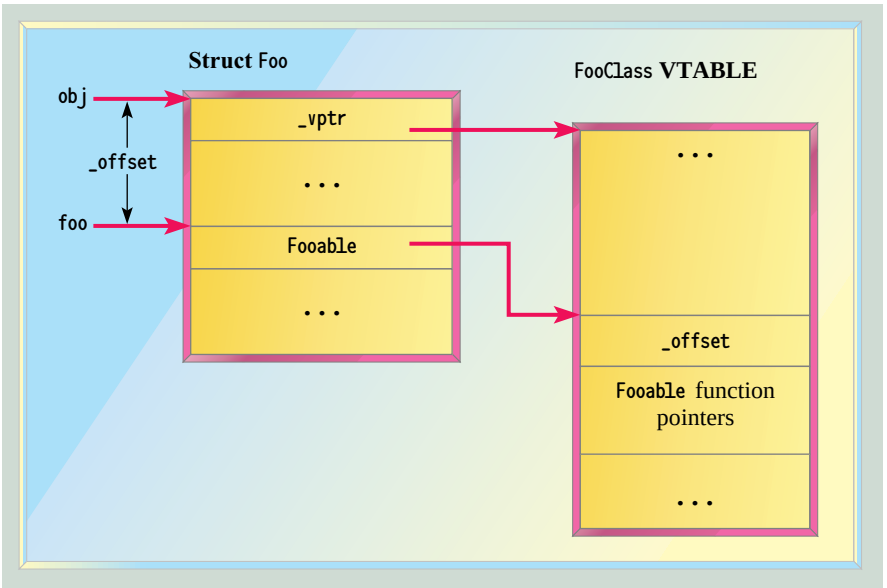
The first step is generated automatically by the macro `BEGIN_VTABLE`. Copying the inherited VTABLE guarantees that adding new virtual functions to the superclass doesn't break any subclasses; in other words, no manual changes to subclasses are necessary (you will only have to recompile the subclass code). Unless a given class explicitly chooses to override the superclass behavior, the inherited implementation is adequate. Of course if a class declares its own virtual functions, the corresponding function pointers will not be initialized during this step.

I also provide the macros `VMETHOD` and `IMETHOD` to facilitate the second step of binding virtual functions to their implementation. If you cannot provide implementation for a given method—you intend it to be a "pure virtual function" implemented by subclasses—you should still initialize the function pointer with an `Object_NoIm` dummy implementation. `Object_NoIm` interrupts execution (through a failing assertion), which helps detecting unimplemented abstract methods at run-time.

As I've mentioned, each object keeps a pointer (virtual pointer) to its class descriptor, inherited from the object class. The virtual pointer needs to be set up correctly during object initialization, that is, in the constructor. This must be done after the superclass constructor call because the superclass constructor will set this pointer to point to the superclass VTABLE. If the VTABLE for the object being initialized isn't set up yet, the VTABLE constructor should be called. These two steps are accomplished by invoking the macro `VHOOK`. Note that by the time the superclass constructor has done the same with the superclass VTABLE, so the whole class hierarchy is initialized properly.

Listing 2 shows the definition of the class `String` declared in Listing 1. The VTABLE of this class overrides only the virtual destructor implementation. I used explicit upcasting of the `this` argument of the destructor before assigning it to `VMETHOD (Object, Des)` to avoid compiler warnings. Note also how the constructor first calls the superclass constructor, then hooks the virtual pointer, and finally initializes attributes.

FIGURE 2
Memory layout of class `Foo` *implementing interface* `Fooable`. *Pointer* `foo` *allows you to access the* `Fooable VTABLE (*(*foo))` *as well as to reconstruct the pointer* `obj` *to the class attribute* `struct (obj = (Object)((char*)foo - (*foo) -> _offset)`.



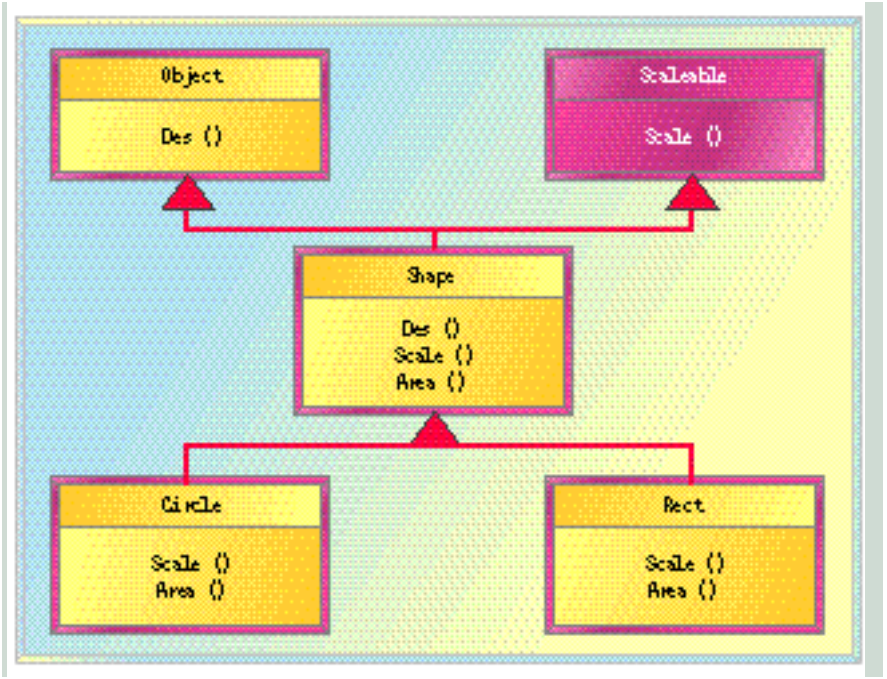
INTERFACES

Sometimes you need only to define abstract methods that an object must support, but not necessarily commit to a specific implementation. Manipulating objects solely in terms of the interfaces greatly reduces implementation dependencies between subsystems, so that one of the major principles of reusable OO design is: "program to interface, not to implementation."[4] Java addresses this design need very elegantly with support for interfaces (Gosling, 1995 and Arnold, 1996).[5]

My approach to Java-style interfaces in C is just a generalization of the VTABLE concept. What if a class defined only abstract methods (purely virtual functions) but did not define any attributes? Such a class would be represented only by its VTABLE. Inheriting from this class would only require maintaining a virtual pointer to the corresponding VTABLE and of course, implementing all abstract methods. An object could easily maintain many such pointers, so multiple inheritance from such specific classes (interfaces) would be simple.

This doesn't completely solve the memory alignment problem. You cannot simply use a pointer to an object in a place where an interface is expected, because there is no way of finding the corresponding VTABLE. This is because the additional virtual pointers cannot be aligned with the virtual pointer `__vptr` inherited from `Object`.

FIGURE 3
Class diagram of Shape *and its subclasses. (Interface box is pink.)*



LISTING 2
Definition of class String.

```
#include <stdlib.h>
#include <string.h>
#include "string.h"
/** implementation of String class */
BEGIN_VTABLE(String, Object)
   VMETHOD(Object, Des) = (void (*)(Object))StringDes;
END_VTABLE
String StringCon1(String this, const char *str) {
   Object_Con(&this->super); /* construct superclass */
   VHOOK(this, String);   /* hook String VPTR */
   /* allocate and initialize the buffer */
   this->__buffer = (char*)malloc(strlen(str) + 1);
   if (!this->__buffer)
      return NULL;      /* failure */
   strcpy(this->__buffer, str);
   return this;
}
String StringCon2(String this, String other) {
   return StringCon1(this, StringToChar(other));
}
const char *StringToChar(String this) {
   return this->__buffer;
}
void StringDes(String this) {
   free(this->__buffer);   /* release buffer */
   Object_Des(&this->super); /* destroy superclass */
}
```

As always in this kind of situation, you can use an additional level of indirection to solve the problem. As shown in Figure 3, a pointer `foo` to a `Fooable` VPTR located inside an object that implements interface `Fooable` is a good starting point. This pointer can be used both to access the `Fooable` virtual table (to resolve the virtual call) and to reconstruct the location of the object (to provide the `this` argument).

Interfaces differ significantly from classes—they don't descend from `Object` (because they define no attributes) and they define different VTABLEs containing the `__offset` field. Note that virtual functions defined by interfaces also must use the `this` pointer, but it should be of generic `Object` type. The sample code I discuss in the next section demonstrates how you can use interfaces to write code without committing to any particular implementation.

SAMPLE CODE

To illustrate the concepts I've discussed, I've provided an implementation of the simple class hierarchy depicted in Figure 3. (Complete listings are provided on the *ESP* Web site, *www.embedded.com/code*). Class `Shape` extends `Object` and implements the `Scalable` interface. This is an abstract class (designed for inheritance only), so it protects its constructor and destructor. Class `Shape` contains the `String` object as its member, demonstrating object composition. The `Scalable` interface defines only one abstract method, `Scale()`. Concrete classes `Circle` and `Rect` both extend `Shape` and override `Scale()` and `Area()` methods. The test case allocates the `Circle` object on the stack frame and array of `Rect` objects on the heap. Test functions for the `Shape` class and `Scalable` interface demonstrate dynamic binding for classes and interfaces, respectively.

As an exercise, you can modify the code by adding attributes or virtual functions to the `Shape` class (or even the `Object` base class). You can convince

yourself that these modifications don't require any manual changes to subclasses. I also strongly recommend that you step through the code using a debugger.

WHAT'S TO LOSE?

So what do you lose by using C rather than an OO language? Using the techniques I've presented here, you don't have to sacrifice much convenience and expressiveness, because you can fairly easily map most important OO concepts to C. You don't have to compromise much maintainability either, because you can automate many tasks. The most important feature of this implementation is that adding new attributes and methods (including virtual functions and interfaces) to the superclass doesn't require any manual changes to subclasses.

The real issue is that C requires significantly higher programming discipline in object initialization and cleanup than OO languages, especially those with garbage collection. But this is a well known deficiency of C, which isn't to be repaired easily. (For example, C++ is still plagued by most of these problems.)

Encapsulation, inheritance, and polymorphism are nothing but design patterns at the C language level.

Encapsulation, inheritance, and polymorphism are nothing but design patterns at the C language level (Gamma, 1995). As do all design patterns, they bring a higher (OO) level of abstraction by introducing their specific naming conventions and idioms. If you aren't comfortable with my conventions, you should adopt your own. Most important is consistency, which dramatically improves code readability and allows for quick identification of the patterns.

If you start using these patterns, you will probably notice that they completely change your thinking and programming style in C. You'll not only find your C code to be more similar to Java, you'll find yourself more prepared for Java. ESP

Miro Samek is a software engineer at GE Medical Systems, currently developing real-time, embedded software for diagnostics x-ray equipment. He holds a PhD in physics from Jagiellonian University in Cracow, Poland. Miro has previously worked on nuclear physics experiments at GSI Darmstadt, Germany. He may be reached at miroslaw.samek@med.ge.com.

REFERENCES

1. Gosling, James and Henry McGilton, "The Java Language Environment: A White Paper," Sun Microsystems, 1995.

2. Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Loernsen. *Object-Oriented Modelling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

3. Eckel, Bruce. *Thinking in C++*. Englewood Cliffs, NJ: Prentice Hall, 1995.

4. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

5. Arnold, Ken and James Gosling. *The Java Programming Language*. Reading, MA: Addison-Wesley, 1996.