Ficl - Object Forth Wraps C Structures

20th FORML Conference
1 November 1998
John Sadler
john_sadler@alum.mit.edu

Abstract

Ficl is an ANS Forth written in C, designed specifically to be embedded in programs written in C. Ficl combines a simple way to import C functions with a novel object oriented syntax that is capable of wrapping data structures compiled in other languages. In so doing, Ficl can act as an object oriented front-end for small systems written in either Forth or C.

Introduction: Why Another Freeware Forth?

Why write another freeware Forth? There are large numbers of them available on the FIG web site, and more in Europe and elsewhere. Several are written in C, and some sport extensive Windows programming capabilities, object oriented extensions, and other syntactical goodies that help with larger coding projects.

I suppose that there are several reasons I wrote Ficl. First, I've had some difficulty getting C programmers to embrace Forth even as an adjunct to their regular coding in C or C++. The last Forth interpreter I used in a product had a kernel written in 68K assembly language. I understood it quite well, but everyone else thought it was ugly to maintain. This is a reasonable complaint in the sense that conservative engineers want to have a good mental model of the tools they use.

To address this problem, I wrote Ficl in ANSI C, and designed it to minimize the porting requirements to systems that have C runtime environments. You can port Ficl to any 32 bit microprocessor by taking a few minutes to inspect the main header file, possibly redefining some simple typedefs, and by filling in three function stubs. Two of these map very closely to the Standard C functions *malloc* and *free*. The third gets text out of the interpreter – this solves a common problem in mapping the C runtime to small systems: no Standard I/O or file system.

While lots of programmers know C, this language is not optimally suited to prototyping and rapid development for reasons that will be obvious to experienced Forth programmers: the time and typing overhead it takes to create even a simple program. There is clearly an opportunity for a programming environment like Forth even in organizations where C or C++ are the intended production language. Ficl attempts to address this niche by being very straightforward to integrate into C programs while requiring minimal system resources. Ficl allows C functions to be exported to the interpreter through a programming interface that does not require editing the Ficl sources. It also provides object oriented programming extensions that work well with C programs. This last feature might provide a low-overhead path to OOP for

embedded systems developers who find the barriers to entry of C++ too daunting.

One reason for using a standard language rather than inventing a new one is that it should be possible to find books that explain the language to new users, rather than having to write one. Further, the time invested in learning a standard language seems more likely to be repaid by future applications. That's why Ficl conforms to the Forth DPANS (and presumably the ANS as well).

Taken together, here's the list of requirements I wrote when starting to design Ficl [6]:

- Scripting, prototyping, and extension language for programs written in C or C++
- Target 32 bit microprocessors with a C runtime environment
- Minimize porting overhead given the above constraints
- Conform to the DPANS, providing the CORE word set plus SEARCH and LOCAL optional word sets
- Provide object oriented programming extensions
- Make the code as transparent as possible
- Provide a means to export C functions to the interpreter

My web searches turned up nothing that met these goals, so here it is. One additional reason for addressing the scripting/prototyping niche is my hope that Ficl will be a "stealth" Forth – that people might adopt Ficl for its ease of integration, and be drawn to Forth once they realize that it's much more than a scripting language. We'll see.

Ficl's C/Forth Interface

Many other Forths written in C have large switch statements at the core of their inner interpreters. In this model, primitives are assigned small integer tokens. To add a Forth wrapper to application specific code written in C, you have to add a new case to the switch for each function to be exported. This forces external code interfaces to be concentrated in the token interpreter rather than co-located with functionally related code. This presents maintenance problems in most cases.

We can remedy that situation by adding a Forth primitive that calls a C function indirectly through a stored address, similar to the way **DOES>** vectors to code defined in another word. This might permit an interface from Forth to C: discover the address of some function, execute Forth code to create a word and cause the new word to call the C function when executed. This has the disadvantage that it's not automatic to know the address of a C function from Forth.

The Forth interpreter can fix this with an interface function that host programs can call to bind application domain functions to words in the dictionary. If we require that all C functions exported to our Forth have the same signature (number and type of parameters, and return type), then the builder function

would need only the address of the target function, and the name to bind in the dictionary. The builder would append a definition to the dictionary that calls the target C function supplying whatever parameters the convention specifies. Here's a sample prototype for such a builder:

```
void ficlBuild(char *name, void *function);
```

An alternative is to allow the interface to specify the number of cells to pop off the parameter stack and push onto the C call stack before invoking the C interface function. This makes it possible to wrap a broader range of C functions in Forth words, as long as all their parameters are properly aligned. It requires the builder to provide a means to specify the number and width of parameters to push on the C calling stack, and the size of the return value, if any. The simplest way to do this is to require that all interface functions have parameters of one specified width. The builder function might then look like this:

```
void ficlBuild(char *name, void *function, int nParams, int
nReturn);
```

Otherwise, there needs to be a more complex protocol for specifying each parameter's width. The variable parameter designs require information about the function that the C compiler neither supplies nor checks for consistency. On the other hand, the constant signature design requires a special wrapper function to be written for each exported word. The wrapper function's job is to marshal parameters from Forth to C explicitly. An advantage of this approach is that the compiler can check that the target function call has the correct number and widths of parameters. Ficl uses this wrapper function technique to import C functions.

Now that there is a way to import functions written in C to our Forth, why not get rid of the switch statement and write all of the primitives this way? Ficl does this, reducing the inner interpreter to a small loop. Now application-specific words have exactly the same execution mechanics as any other Forth primitive.

The wrapper function technique generally requires that interface functions be written explicitly for Ficl. This is not hard. Ficl's interface builder really is called ficlBuild(), and it expects three parameters: the name of the word to be created, a pointer to a function to execute, and a bit-field that specifies IMMEDIATE and compile-only attributes. When the wrapper function executes, it gets as its one parameter a pointer to the Ficl virtual machine that's executing it, and it returns nothing. The wrapper function can use the virtual machine pointer to manipulate the stacks and the input buffer.

Ficl provides public functions to push and pop stacks, perform run-time stack depth checking, and manipulate the dictionary. A typical wrapper function pops some parameters off Ficl's stack, passes them to a C function, and pushes the result onto Ficl's stack again.

Host applications get text to Ficl by calling ficlExec. This function causes a virtual machine to execute a chunk of text. FiclExec is reentrant, so wrapper functions can use it too. This is another distinction between Ficl and other Forths written in C: Ficl's outer interpreter does not expect to get more input text from

any specific place – it just returns control to the host application when it gets hungry.

Ficl Object Goals

Back on the Web, I started looking for established practice in Object extensions for Forth. It appears that most Forth object extensions [4,7] model their internals after C++ in the sense that each class contains a pointer table (a *vtable* in C++ jargon) that maps small integer messages to execution tokens that are the corresponding methods of the class. In order to create a class, you must first create a message map that has a slot for each method of the class. Each class contains a pointer to the message map, among other things. This appears to impose a couple of unpleasant restrictions: derived classes use the same message maps as their parents, so a derived class cannot add methods, it can only override methods defined in its parent class. In addition, the words that represent messages to the class are in a public wordlist. It's possible for two unrelated classes to define the same message with different values.

I wanted Ficl to have a simple and flexible object model that emphasized what I saw as the two original motives for developing OO programming in the first place: safety and reuse. Safety in this case means to make sure that data and the operations that are defined on the data are always matched. Inheritance provides the reuse mechanism.

Classic definitions of object oriented programming [1,2] list the three essential attributes of an OO language as encapsulation, polymorphism, and inheritance. We mentioned encapsulation and inheritance in the previous paragraph. Polymorphism, the mapping of a particular message to different methods depending on the class of the receiver, implies late binding. In order to be safe, I decided to make late binding Ficl's default behavior, and to provide early binding upon request for efficiency. Late binding guarantees that the appropriate method will be invoked for a given message, while early binding can cause misunderstandings.

In order to realize the design goal of full interoperation between Ficl and its host program, I added the requirement that Ficl's object model be somehow capable of acting as an adapter, to model data structures written in C. Here's the list of Ficl Object design goals [6]:

- Ficl objects are normally late bound for safety (late binding guarantees that the appropriate method will always be invoked for a particular object). Early binding is also available, provided you know the object's class at compile-time.
- Support single inheritance, aggregation, and arrays of objects.
- Classes have independent name spaces for their methods: methods are only visible in the context of a class or object.
- Methods can be overridden or added in subclasses
- No fixed limit on the number of methods of a class or subclass

- Ficl OOP syntax is regular and unified over classes and objects. In Ficl, classes are objects. Class methods include the ability to subclass and instantiate.
- Adapt legacy data structures with object wrappers. You can model a structure in a Ficl class, and create an instance that refers to an address in memory that holds an instance of the structure. The *ref object* can manipulate the structure directly. This lets you wrap data structures written and instantiated in C.
- Be thread-safe so that concurrent virtual machines can use objects

Ficl Object Theory

The most significant departure Ficl takes from other OO Forths [4,7] is that Ficl represents an object as a cell pair on the stack. One cell points to the instance data, and the other points to the class:

```
( instance-addr class-addr )
```

Whenever a named Ficl object executes, it leaves this "signature". All methods expect a class and instance on the stack when they execute, too. In many other OO languages, including C++, instances contain information about their classes (a vtable pointer, for example). By making this pairing explicit rather than implicit, Ficl can be OO about chunks of data that don't realize that they are objects, without sacrificing any robustness for native objects. Whenever you create an object in Ficl, you specify its class. After that, the object always pushes its class and the address of its payload when invoked by name. To wrap some external data structure with an object model, you first create the class that models the structure, then tell the class to make a *ref instance* of itself, supplying the address of the data structure as a parameter. The new ref instance behaves as if it is a native Ficl object.

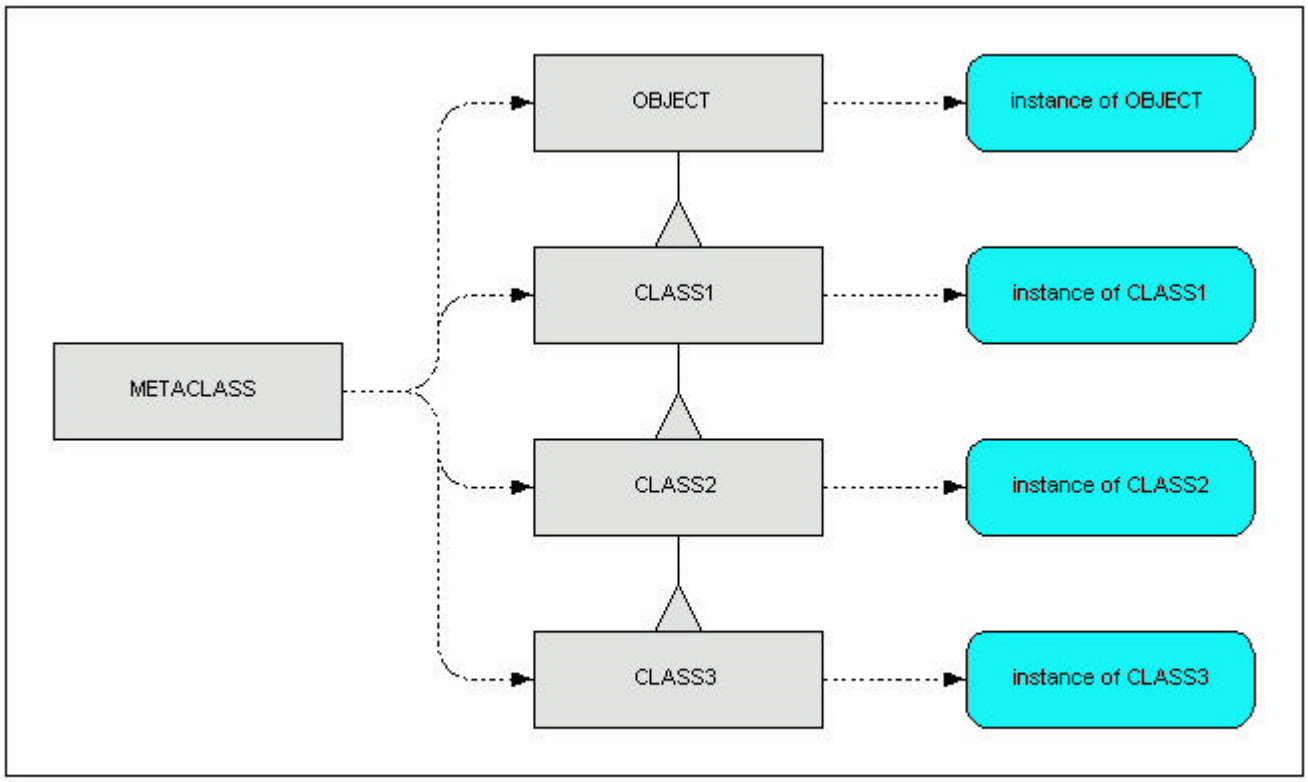
Classes are special kinds of objects that store the methods of their instances, the size of an instance's payload, and a parent class pointer. Classes themselves are instances of a special base class called METACLASS, and all classes inherit from class OBJECT. This results in a very simple syntax for constructing and using objects. Class methods include subclassing (SUB), creating initialized and uninitialized instances (NEW and INSTANCE), and creating reference instances (REF). Classes also have methods for disassembling their methods (SEE), identifying themselves (ID), and listing their pedigree (PEDIGREE). All objects inherit methods for initializing instances and arrays of instances, for performing array operations, and for getting information about themselves.

All classes in Ficl *derive* from the common base class OBJECT. All classes are *instances* of METACLASS. This means that classes are objects, too. METACLASS implements the methods for messages sent to classes. Class methods create instances and subclasses, and give information about the class. Classes have exactly three elements:

- The address (.CLASS) of a parent class, or zero if it's a base class (only OBJECT and METACLASS have this property)
- The size (.SIZE) in address units of an instance of the class
- A wordlist ID (.WID) for the methods of the class

In the figure below, METACLASS and OBJECT are system-supplied classes. The others are contrived to illustrate the relationships among derived classes, instances, and the two system base classes. The dashed line with an arrow at the end indicates that the object/class at the arrow end is an instance of the class at the other end. The vertical line with a triangle denotes inheritance.

Note for the curious: METACLASS behaves like a class - it responds to class messages and has the same properties as any other class. If you want to twist your brain in knots, you can think of METACLASS as an instance of itself.



The Ficl Object Model

All classes are instances of `METACLASS` and subclasses of `OBJECT`

Ficl OOP Syntax

The most important Ficl OOP word is -->. This word, which I pronounce late-bind for lack of a better name, binds a message to an object:

```
politician --> get-tough-on-something  \ it's an election year
```

There is an early bind operator too. It's only defined while compiling, and since it binds early, its symbol is shorter: =>. To use early binding, you must specify a class and message. Early-bind finds the corresponding method and compiles its execution token into the current definition. To make this process a bit cleaner syntactically, Ficl classes and the early-bind operator are IMMEDIATE.

To create a Ficl class, you have to subclass OBJECT or some other class that has OBJECT at the base of its pedigree. Any class can respond to the SUB method, creating a subclass of itself:

```
bourgeoisie --> sub proletariat
```

Having done that, you next define instance variables and methods. Instance variables are declared using the structure syntax of John Hayes [4]. Instance variable builder words reserve space in the class for members, and create methods that push the offsets of their member variables when executed. You can also override methods and define new methods simply by making colon definitions in the context of the class. To end a class context, you invoke END-CLASS. Class definitions can be nested to create data structures that refer to each other – partially complete class definitions are visible in the dictionary.

Here's an example of an actual Ficl class definition:

```
object --> sub c-ref
    cell: .class
    cell: .instance
      : get   ( inst class -- refinst refclass )
            drop 2@ ;
      : set   ( refinst refclass inst class -- )
            drop 2! ;
end-class
```

This class stores a pointer to an object. It has two member variables: the address of instance variables and the address of the class that describes them. The get and set methods do the obvious – they perform size-appropriate fetch and store for the instance variables.

To create an instance of c-ref, we can send the new class another message:

```
c-ref --> new ref-instance
```

The c-ref class makes an instance of itself named ref-instance, and clears it. Storage for the new instance comes from the dictionary – Ficl does not yet support allocation from a heap.

For complete information and a tutorial on Ficl's OO syntax, please see the Ficl release notes [6]

Wrapping C Structures

We can use the OOP facilities to wrap data structures in C once we create some simple base classes that model scalar data types of C. Ficl provides several of these, including 1, 2, and 4 byte scalar quantities and pointers to them. Here's an example from the Ficl sources of a C structure and its Ficl object model:

In C:

```
/*
** Ficl models memory as a contiguous space divided into
** words in a linked list called the dictionary.
** A FICL_WORD starts each entry in the list.
** Version 1.02: space for the name characters is allotted from
** the dictionary before the word header
*/
typedef struct ficl_word
{
    struct ficl_word *link;  /* Previous word in the dictionary    */
    UNS16 hash;
    UNS8 flags;              /* Immediate, Smudge, Compile-only    */
    FICL_COUNT nName;        /* Number of chars in word name       */
    char *name;              /* First nFICLNAME chars of word name */
    FICL_CODE code;          /* Native code to execute the word    */
    CELL param[1];           /* First data cell of the word        */
} FICL_WORD;
```

Now in Ficl's OO Forth:

```
object subclass[1] c-word
    c-word     ref: .link
    c-2byte    obj: .hashcode
    c-byte     obj: .flags
    c-byte     obj: .nName
    c-bytePtr  obj: .pName
    c-cellPtr  obj: .pCode
    c-4byte    obj: .param0
    \ Push word's name...
    : get-name   ( inst class -- c-addr u )
        2dup
        --> .pName --> get-ptr  -rot
        --> .nName --> get
    ;
    : next   ( inst class – link-inst class )
        --> .link ;
    : info
        ." ficl word: "
        2dup --> get-name type cr
        ." hash = "
        2dup --> .hashcode --> get x. cr
        ." flags = "
        --> .flags --> get x. cr
    ;
end-class
```

The first line creates `c-word`, a class that inherits directly from `object`. The next line is the first member variable declaration. It declares `.link` as a *ref* to c-word. A ref member is a pointer to an object of fixed class. in this case, .link always pushes c-word in the class part of its signature when invoked. There is no way to assign a value to a ref member variable – this facility is only useful for modeling pre-initialized structures. If we wanted to create a native Ficl class with a member that pointed to another object, we could simply incorporate a member

[1] `subclass` is equivalent to `--> sub`

variable of type c-ref instead, and have the ability to use its set and get methods as shown earlier.

The next six lines tile various scalar instance variables into the class. We have to know how the C compiler will pad structures, and how much space it allots for each of the scalar types. In this case, I've assumed that the compiler packs as tightly as possible. By the way, all Ficl internal structures are designed to even-align double byte members and quad-align quad-byte members. This should help keep the member offsets invariant over compiler alignment behavior (but it's not a guarantee).

Following the member variable declarations (by convention, not by requirement) are some simple method definitions. `Get-name` pushes the `( c-addr u )` representation of the word's name. It uses the `get` method twice – first to get the address, and next to get the count. It's handy to have `–rot` in this context – I like to think of it as `one-and-a-half-swap` because it has the effect of swapping a single cell on top of the stack with a cell-pair underneath. There is no "self" or "this" variable in Ficl OOP – objects get manipulated only via the stack. This simplifies the syntax quite a bit, at the cost of some extra stack twiddling. (Stack twiddling builds character in Forth programmers.)

The method called `next` follows the word's link to the next one in the dictionary link list. As you can see from the source, it's just an alias for `.link`.

Now that we have a model for a Ficl word, we can wrap an actual word with it by making a ref instance that refers to the word:

```
' c-ptr  c-word --> ref ptrref
```

Makes a ref called ptrref that thinks it's the word c-ptr…

```
ptrref --> get-name type cr  \ prints "c-ptr"
ptrref --> next --> get-name type cr  \ prints "c-2byte"
```

The FICL_WORD struct defined above in C makes no concession to being object oriented. Ficl can model such C structures and make them appear as useful objects in Forth. As is typical with object extensions to Forth, Ficl OOP took about 200 lines of code to create. Perhaps Forth is the "better C" for embedded systems.

Future Directions

Mainstream OO languages and C++ especially have fallen short of the goal of permitting reuse of software components in a vendor and platform independent way. Distributed object models such as COM and CORBA attempt to address this situation. Both models claim to provide a vendor neutral, platform and location independent protocol for software components to interoperate. Many mainstream programming languages have COM and CORBA mappings. I'd like to add this capability (at least a COM mapping) to Ficl.

It would be helpful to automate the generation of Ficl wrapper functions. There exist programs [5] designed to do this task for scripting languages such as Perl

and Python. A port of such a program, especially if it could also generate Ficl object wrapper code, would be a helpful adjunct to Ficl OOP.

Finally, I'd like to use Ficl to create some embedded control applications that have been on the back burner for several years now.

References

1. Smalltalk-80: the Language – Adele Goldberg, Dave Robson – Addison Wesley, 1989
2. Object Oriented Software Systems – David Robson – *Byte*, August 1981
3. Portable Inheritance and Polymorphism in C – Miro Samek – *Embedded Systems Programming*, December 1997
4. Objects for Small Systems – John Hayes – *Embedded Systems Programming*, March 1992
5. SWIG and Automated C/C++ Scripting Extensions – David Beazley – *Dr. Dobb's Journal*, February, 1998
6. Ficl 2.02 Release Notes – John Sadler – http://www.taygeta.com/ficl.html
7. Yet another Forth objects package – M. Anton Ertl – http://www.complang.tuwien.ac.at/forth/objects/objects.html

Acknowledgements

Thanks to my wife Laura for enduring my mysterious late night coding frenzies, to Skip Carter for hosting Ficl on the Taygeta web site, and to those who have used Ficl and lived to tell the tale.