Q: My Objective-C messages take too long compared with C function calls. What can I do?

A: We have chosen to pay a certain performance cost in exchange for the benefits of object-oriented programming. What is really important is to amortize the added overhead of a message send over the cost of running your program. The cost of writing and maintaining correct code is also a consideration.

There is a way to optimize tight loops in code. The method "methodFor:" returns a pointer to the actual function used to implement a method. This can be used at the beginning of a tight loop to cache the call. Example:

```
IMP func;

func = [anObject methodFor:aSelector];
while (loopingTightly) {
    ...
    func(anObject, @selector(aSelector), args ...);
    ...
}
```

As long as anObject is not changing value in that inner loop, then this technique does not sacrifice any of the advantages of Objective-C (e.g., dynamic binding) and is just as efficient as normal C.

However, there is a very subtle error that can be introduced by using the methodFor: method to get the

pointer to the implementation of a method. methodFor: returns an IMP which is a pointer to a function which returns an id. It is prototyped to take an id, a SEL, and then a variable number of arguments, which are, of course, unprototyped because depending on the method, those arguments could be anything. This lack of a complete ANSI prototype can produce ill effects. Without prototypes, arguments are subject to promotion. Chars are promoted to int, floats are promoted to double, and so on.

The concern is with floats. Let's say you have a method that takes a float as an argument. Objective-C methods conform to the ANSI prototyping rules and expect a 32-bit float to be put on the stack for that method. And when that method is called normally, 32 bits are put on the stack and it all works out.

Now you get the IMP by calling methodFor: for that selector. You try to directly call the function, passing a float. Because the function was not explicitly prototyped, old-style C takes over and that float is promoted to a 64-bit double. When that 64 bits is placed on the stack for a method which expects a 32-bit float Ð catastrophe strikes! The method tries to interpret the first 32 bits of the 64-bit double as a float, which doesn't work, and any arguments which follow the float in the parameter list are also messed up.

The solution: construct the proper prototype for the function pointer in question , something like this:

```
 id   (*func)(id,SEL,float,...);

func = (id (*)(id,SEL,float,...))[anObject methodFor:@selector(aSelector)];
```

```
(* func)(anObject, @selector(aSelector), aFloat, otherArgs ...);
```

The prototype ensures that methodFor: returns a pointer to a function that returns an id and has the arguments an id, a selector, a float, and other variable arguments. If the sight of prototypes scares you, another solution is to structure the necessary methods to simply take doubles.

You must take similar precautions and coerce the return value when the method does not return an id. Here floatValue is a method which returns a float.

```
 float   (*getFloat)();
float    value;

getFloat = (float (*)())[anObject methodFor:@selector(floatValue)];
value = (* getFloat)(anObject, @selector(floatValue)));
```

QA219

Valid for 1.0, 2.0, 3.0