How Non-Member Functions Improve Encapsulation

I'll start with the punchline: If you're writing a function that can be implemented as either a member or as a non-friend non-member, you should prefer to implement it as a non-member function. That decision *increases* class encapsulation. When you think encapsulation, you should think non-member functions.

Surprised? Read on.

Background

When I wrote the first edition of *Effective C++* in 1991, I examined the problem of determining where to declare a function that was related to a class. Given a class `C` and a function `f` related to `C`, I developed the following algorithm:

```
1  if (f needs to be virtual)
2     make f a member function of C;
3  else if (f is operator>> or
4           operator<<)
5     {
6     make f a non-member function;
7     if (f needs access to non-public
8         members of C)
9        make f a friend of C;
10    }
11 else if (f needs type conversions
12          on its left-most argument)
13    {
14    make f a non-member function;
15    if (f needs access to
16        non-public members of C)
17
```

```
18      make f a friend of C;
19    }
20  else
      make f a member function of C;
```

This algorithm served me well through the years, and when I revised *Effective C++* for its second edition in 1997, I made no changes to this part of the book.

In 1998, however, I gave a presentation at Actel, where Arun Kundu observed that my algorithm dictated that functions should be member functions even when they could be implemented as non-members that used only `C`'s public interface. Is that really what I meant, he asked me? In other words, if `f` could be implemented as a member function or a non-friend non-member function, did I really advocate making it a member function? I thought about it for a moment, and I decided that that was not what I meant. I therefore modified the algorithm to look like this:

```
1  if (f needs to be virtual)
2    make f a member function of C;
3  else if (f is operator>> or
4           operator<<)
5    {
6    make f a non-member function;
7    if (f needs access to non-public
8        members of C)
9      make f a friend of C;
10   }
11 else if (f needs type conversions
12          on its left-most argument)
13   {
14   make f a non-member function;
15   if (f needs access to non-public
```

```
16        members of C)
17      make f a friend of C;
18   }
19else if (f can be implemented via C's
20          public interface)
21   make f a non-member function;
22else
23   make f a member function of C;
```

Since then, I've been battling programmers who've taken to heart the lesson that being object-oriented means putting functions inside the classes containing the data on which the functions operate. After all, they tell me, that's what encapsulation is all about.

They are mistaken.

Encapsulation

Encapsulation is a *means*, not an end. There's nothing inherently desirable about encapsulation. Encapsulation is useful only because it yields other things in our software that we care about. In particular, it yields flexibility and robustness. Consider this struct, whose implementation I think we'll all agree is unencapsulated:

```
1struct Point {
2   int x, y;
3};
```

The weakness of this struct is that it's not flexible in the face of change. Once clients started using this struct, it would, practically speaking, be very hard to change it; too much client code would be broken. If we later decided we wanted to compute `x` and `y` instead of storing those values, we'd probably be out of luck. We'd be similarly thwarted if we decided a superior design would be to look `x` and `y` up in a database. This is the real problem with poor encapsulation: it precludes future implementation changes. Unencapsulated software is inflexible, and as a result, it's not very robust. When the world changes, the software is unable to gracefully change with it. (Re-

member that we're talking here about what is *practical*, not what is possible. It's clearly possible to change `struct Point,` but if enough code is dependent on it in its current form, it's not practical.)

Now consider a class with an interface that offers clients capabilities similar to those afforded by the struct above, but with an encapsulated implementation:

```
1 class Point {
2 public:
3    int getXValue() const;
4    int getYValue() const;
5    void setXValue(int newXValue);
6    void setYValue(int newYValue);
7
8 private:
9   ...                 // whatever...
10};
```

This interface supports the implementation used by the struct (storing `x` and `y` as `int`s), but it also affords alternative implementations, such as those based on computation or database lookup. This is a more flexible design, and the flexibility makes the resulting software more robust. If the class's implementation is found lacking, it can be changed without requiring changes to client code. Assuming the declarations of the public member functions remain unchanged, client source code is unaffected. (If a suitable implementation has been adopted, clients need not even recompile.)

Encapsulated software is more flexible than unencapsulated software, and, all other things being equal, that flexibility makes it the superior design choice.

Degrees of Encapsulation

The class above doesn't fully encapsulate its implementation. If the implementation changes, there's still code that might break. In particular, the member functions of the class might break. In all likelihood, they are dependent on the particulars of the data members of the class. Still, it seems clear that the class is more encapsulated than the

struct, and we'd like to have a way to state this more formally.

It's easily done. The reason the class is more encapsulated than the struct is that more code might be broken if the (public) data members in the struct change than if the (private) data members of the class change. This leads to a reasonable approach to evaluating the relative encapsulations of two implementations: if changing one might lead to more broken code than would the corresponding change to the other, the former is less encapsulated than the latter. This definition is consistent with our intuition that if making a change is likely to break a lot of code, we're less likely to make that change than we would be to make a different change that affected less code. There is a direct relationship between encapsulation (how much code might be broken if something changes) and practical flexibility (the likelihood that we'll make a particular change).

An easy way to measure how much code might be broken is to count the functions that might be affected. That is, if changing one implementation leads to more potentially broken functions than does changing another implementation, the first implementation is less encapsulated than the second. If we apply this reasoning to the struct above, we see that changing its data members may break an unknowably large number of functions — every function that uses the struct. In general, we can't count how many functions this is, because there's no way to locate all the code that uses a particular struct. This is especially true for library code. However, the number of functions that might be broken if the class's data members change is easy to determine: it's all the functions that have access to the private part of the class. That's just four functions (assuming none are declared in the private part of the class), and we know that because they're all conveniently listed in the class definition. Since they're the only functions that have access to the private parts of the class, they're the only functions that can be affected if those parts change.

Encapsulation and Non-Member Functions

We've now seen that a reasonable way to gauge the amount of encapsulation in a class is to count the number of functions that might be broken if the class's implementation changes. That being the case, it becomes clear that a class with n member functions is more encapsulated than a class with n+1 member functions. And that observation is what justifies my argument for preferring non-member non-friend functions to member functions: if a function `f` could be implemented as a member function or

as a non-friend non-member function, making it a member would decrease encapsulation, while making it a non-member wouldn't. Since functionality is not at issue here (the functionality of `f` is available to class clients regardless of where `f` is located), we naturally prefer the more encapsulated design.

It's important that we're trying to choose between member functions and *non-friend non-member* functions. Just like member functions, friend functions may be broken when a class's implementation changes, so the choice between member functions and friend functions is properly made on behavioral grounds. Furthermore, we now see that the common claim that "friend functions violate encapsulation" is not quite true. Friends don't violate encapsulation, they just decrease it — in exactly the same manner as member functions.

This analysis applies to any kind of member functions, including static ones. Adding a static member function to a class when its functionality could be implemented as a non-friend non-member decreases encapsulation by exactly the same amount as does adding a non-static member function. One implication of this is that it's generally a bad idea to move a free function into a class as a static member just to show that it's related to the class. For example, if I have an abstract base class for `Widgets` and then use a factory function to make it possible for clients to create `Widgets`, the following is a common, but inferior way to organize things:

```
1 // a design less encapsulated than it could be
2 class Widget {
3    ...           // all the Widget stuff; may be
4                  // public, private, or protected
5
6 public:
7
8    // could also be a non-friend non-member
9    static Widget* make(/* params */);
10};
```

A better design is to move `make` out of `Widget,` thus increasing the overall encapsu-

lation of the system. To show that `Widget` and make are related, the proper tool is a namespace:

```
// a more encapsulated design
namespace WidgetStuff {
   class Widget { ... };
   Widget* make( /* params */ );
};
```

Alas, there is a weakness to this design when templates enter the picture.

Templates and Factory Functions at Namespace Scope

In the previous section, I argued that static member functions should be made non-members whenever that is possible, because that increases class encapsulation. I consider these two possible implementations for a factory function:

```
// the less encapsulated design
class Widget {
   ...
public:
   static Widget* make(/* params */);
};

// the more encapsulated design
namespace WidgetStuff {
   class Widget { ... };
   Widget* make( /* params */ );
};
```

Andrew Koenig pointed out that the first design (where `make` is static inside the class) enables one to write a template function that invokes `make` without knowing the type of what is being made:

```
1template<typename T>
2void doSomething( /* params */ )
3{
4   // invoke T's factory function
5   T *pt = T::make( /* params */ );
6   ...
7}
```

This isn't possible with the namespace-based design, because there's no way from inside a template to identify the namespace in which a type parameter is located. That is, there's no way to figure out what `???` is in the pseudocode below:

```
1template<typename T>
2void doSomething( /* params */ )
3{
4   // there's no way to know T's containing namespace!
5   T *pt = ???::make( /* params */ );
6   ...
7}
```

For factory functions and similar functions which can be given uniform names, this means that maximal class encapsulation and maximal template utility are at odds. In such cases, you have to decide which is more important and cater to that. However, for static member functions with class-specific names, the template issue fails to arise, and encapsulation can again assume precedence.

Syntax Issues

If you're like many people with whom I've discussed this issue, you're likely to have reservations about the syntactic implications of my advice that non-friend non-member functions should be preferred to member functions, even if you buy my argument about encapsulation. For example, suppose a class `Wombat` supports the functionality of both eating and sleeping. Further suppose that the eating functionality must be implemented as a member function, but the sleeping functionality could be

implemented as a member or as a non-friend non-member function. If you follow my advice from above, you'd declare things like this:

```
1class Wombat {
2public:
3    void eat(double tonsToEat);
4    ...
5};
6
7void sleep(Wombat& w, double hoursToSnooze);
```

That would lead to a syntactic inconsistency for class clients, because for a `Wombat` `w`, they'd write

to make it eat, but they would write

to make it sleep. Using only member functions, things would look much neater:

```
1class Wombat {
2public:
3    void eat(double tonsToEat);
4    void sleep(double hoursToSnooze);
5    ...
6};
7
8w.eat(.564);
9w.sleep(2.57);
```

Ah, the uniformity of it all! But this uniformity is misleading, because there are more functions in the world than are dreamt of by your philosophy.

To put it bluntly, non-member functions happen. Let us continue with the `Wombat` example. Suppose you write software to model these fetching creatures, and imagine that one of the things you frequently need your `Wombats` to do is sleep for precisely

half an hour. Clearly, you could litter your code with calls to `w.sleep(.5)`, but that would be a lot of `.5`s to type, and at any rate, what if that magic value were to change? There are a number of ways to deal with this issue, but perhaps the simplest is to define a function that encapsulates the details of what you want to do. Assuming you're not the author of `Wombat`, the function will necessarily have to be a non-member, and you'll have to call it as such:

```
1// might be inline, but it doesn't matter
2void nap(Wombat& w) { w.sleep(.5); }
3
4Wombat w;
5...
6nap(w);
```

And there you have it, your dreaded syntactic inconsistency. When you want to feed your wombats, you make member function calls, but when you want them to nap, you make non-member calls.

If you reflect a bit and are honest with yourself, you'll admit that you have this alleged inconsistency with all the nontrivial classes you use, because no class has every function desired by every client. Every client adds at least a few convenience functions of their own, and these functions are always non-members. C++ programers are used to this, and they think nothing of it. Some calls use member syntax, and some use non-member syntax. People just look up which syntax is appropriate for the functions they want to call, then they call them. Life goes on. It goes on especially in the STL portion of the Standard C++ library, where some algorithms are member functions (e.g., `size`), some are non-member functions (e.g., `unique`), and some are both (e.g., `find`). Nobody blinks. Not even you.

Interfaces and Packaging

Herb Sutter has explained that the "interface" to a class (roughly speaking, the functionality provided by the class) includes the non-member functions related to the class, and he's shown that the name lookup rules of C++ support this meaning of "interface." This is wonderful news for my "non-friend non-members are better than members" argument, because it means that the decision to make a class-related func-

tion a non-friend non-member instead of a member need not even change the interface to that class! Moreover, the liberation of the functions in a class's interface from the confines of the class definition leads to some wonderful packaging flexibility that would otherwise be unavailable. In particular, it means that the interface to a class may be split across multiple header files.

Suppose the author of the `Wombat` class discovered that `Wombat` clients often need a number of convenience functions related to eating, sleeping, and breeding. Such convenience functions are by definition not strictly necessary. The same functionality could be obtained via other (albeit more cumbersome) member function calls. As a result, and in accord with my advice in this article, each convenience function should be a non-friend non-member. But suppose the clients of the convenience functions for eating rarely needed the convenience functions for sleeping or breeding. And suppose the clients of the sleeping and breeding convenience functions also rarely needed the convenience functions for eating and, respectively, breeding and sleeping.

Rather than putting all `Wombat`-related functions into a single header file, a preferable design would be to partition the `Wombat` interface across four separate headers, one for core `Wombat` functionality (primarily the class definition), and one each for convenience functions related to eating, sleeping, and breeding. Clients then include only the headers they need. The resulting software is not only clearer, it also contains fewer gratuitous compilation dependencies. This multiple-header approach was adopted for the standard library. The contents of namespace `std` are spread across 50 different headers. Clients `#include` the headers declaring the parts of the library they care about, and they ignore everything else.

In addition, this approach is extensible. When the declarations for the functions making up a class's interface are spread across multiple header files, it becomes natural for clients creating application-specific sets of convenience functions to cluster those functions into a new header file and to `#include` that file as appropriate. In other words, to treat the application-specific convenience functions just like they treat the convenience functions provided by the author of the class. This is as it should be. After all, they're all just convenience functions.

Minimalness and Encapsulation

In *Effective C++,* I argued for class interfaces that are complete and minimal. Such in-

terfaces allow class clients to do anything they might reasonably want to do, but classes contain no more member functions than are absolutely necessary. Adding functions beyond the minimum necessary to let clients get their jobs done, I wrote, decreases the class's comprehensibility and maintainability, plus it increases compilation times for clients. Jack Reeves has written that the addition of member functions beyond those truly required violates the open/closed principle, yields fat class interfaces, and ultimately leads to software rot. That's a fair number of arguments for minimizing the number of member functions in a class, but now we have an additional reason: failure to do so decreases a class's encapsulation.

Of course, a minimal class interface is not necessarily the best interface. I remarked in *Effective C++* that adding functions beyond those truly necessary may be justifiable if it significantly improves the performance of the class, makes the class easier to use, or prevents likely client errors. Based on his work with various string-like classes, Jack Reeves has observed that some functions just don't "feel" right when made non-members, even if they could be non-friend non-members. The "best" interface for a class can be found only by balancing many competing concerns, of which the degree of encapsulation is but one.

Still, the lesson of this article should be clear. Conventional wisdom notwithstanding, use of non-friend non-member functions *improves* a class's encapsulation, and a preference for such functions over member functions makes it easier to design and develop classes with interfaces that are complete and minimal (or close to minimal). Arguments about the naturalness of the resulting calling syntax are generally unfounded, and adoption of a predilection for non-friend non-member functions leads to packaging strategies for a class's interface that minimize client compilation dependencies while maximizing the number of convenience functions available to them.

It's time to abandon the traditional, but inaccurate, ideas of what it means to be object-oriented. Are you a true encapsulation believer? If so, I know you'll embrace non-friend non-member functions with the fervor they deserve.

Acknowledgements

Thanks to Arun Kundu for asking the question that led to this article. Thanks also to Jack Reeves, Herb Sutter, Dave Smallberg, Andrei Alexandrescu, Bruce Eckel, Bjarne Stroustrup, and Andrew Koenig for comments on pre-publication drafts that weren't as good as they should have been. (That's why they were drafts.) Finally, great thanks

to Adela Novak for organizing the C++ seminars in Lucerne (Switzerland) that led to the many hours on planes and trains that allowed me to write the initial draft of this article.

Scott Meyers is a recognized authority on C++; he provides consulting services to clients worldwide. He is the author of Effective C++, Second Edition *(Addison-Wesley, 1998),* More Effective C++ *(Addison-Wesley, 1996), and* Effective C++ CD *(Addison-Wesley, 1999). Scott received his Ph.D. in Computer Science from Brown University in 1993.*