Generic Functions and Pointers to Member Functions

Paul R. Potts

August 2006

People who know me know that I still carry a torch for a wonderfully-designed but little-used programming language called *Dylan*. In my arrogant opinion Dylan took much of the best of Common Lisp and CLOS, regularized and streamlined it, and gave it a syntax more acceptable (due to familiarity) to programmers from the C and C++ world (although this annoyed some prefix syntax fans in the process). Dylan is a multi-paradigm language, where "everything is an object," designed to support functional styles, object-oriented styles, and procedural styles.

Dylan is still alive and kicking with several impressive implementations available here including **d2c**, an amazing compiler that generates C code from Dylan source, and which is written primarily in Dylan itself, and the formerly commercial Functional Objects compiler, which generates native machine code. I have not studied the Functional Objects codebase very much, but I have spent time with **d2c**. It is an amazing piece of code. Studying it makes me wish I was able to strap on an auxilliary brain or two and take a pill that gave me a post-graduate education in language implementation without all that tedious mucking about in graduate school, so I could contribute something *useful* to the project instead of just an occasional bug report or ignorant question.

In Dylan, and in Dylan's sire Common Lisp with CLOS, classes contain *data*. Classes don't contain methods. Methods are implemented using a construct known as a *generic function*. A generic function lets you define several *specialized* functions to operate on objects of different classes. A variety of functions with the same name are bundled together; as you define additional functions, they get added to the generic function. At run-time, when you make a call to the generic function, the *specific* function chosen can depend on the *actual* run-time type (the class) of the object you send to it. Classes can inherit from one or more other classes; this inheritance tree is used to make the decision about which actual method to call for a given object. Generic functions can be specialized in other ways; they are extraordinarily flexible and powerful. But I'm only going to touch today on generic function dispatch specialized on the class of the incoming object, incoming parameters, and return type.

In his article "A First Look at Dylan: Classes, Functions, and Modules," Steve Strassmann writes:

> Object-oriented languages, including Dylan, provide polymorphic functions, which means a given function may be executed as one of several possible implementations of that function, called methods… when Dylan sees a call to name(x), depending on what type of object is, one of several methods is selected and executed. In Dylan, name is called a generic function, consisting of a family of name methods that implement the functionality of name for various classes (see Figure 2). Each of these methods "belongs to" its generic function (in this case, name) rather than to a class. This is a key point; it's the core difference between C++'s object model and Dylan's.

In terms of implementation, this means that when you make a generic function call, the generic function isn't a "standard" function; a dispatch mechanism comes into play to select the proper generic function based on the call's arguments. The run-time context contained in the object is *explicitly* passed to the function.

Strassmann gives examples of generic functions; I have stripped down one of his examples somewhat here:

```
// No type declaration, works on any type of object
define method double (x)
   pair(x, x);
end method double;
// Works on all numbers
define method double (x :: &lt;number&gt;)
   2 * x;
end method double;
// Works on all strings
define method double (x :: &lt;string&gt;)
   concatenate(x, x);
end method double;
```

Strassmann writes:

> When double is invoked on an argument, the most specific method is invoked… for example, double("foo") would invoke the third method, because <string> is more specific than <object>, which is what the first method is specialized to. If no match is found, Dylan will catch it and signal an error.

Besides a plethora of different ways to specialize generic functions, Dylan supplies various forms of *introspection*; if you want to extend or alter the run-time dispatch, you can do so.

To C++ programmers, this is inside-out. In C++, classes "contain" methods, known more commonly as "member functions" because they are "members" of the class. C++ supports polymorphism and dispatch based on run-time type.

I'm going to talk a little bit about how objects and member functions work together in C++ and how you can use them, and how, when we look at C++ in light of Dylan or CLOS, C++ actually gives us a *subset* of object dispatch, presented in an *obfuscated* form.

Let's say you don't want to just make a new subclass with specialized methods and allow polymorphic dispatch to operate based on run-time type, but instead you want to handle your own run-time dispatch. In Dylan you can do this using introspection and some handlers that allow you to override the standard generic function dispatch. Since you have the object and the generic function and can get at specific functions, you can just call them, passing the object.

To do the same thing in C++ you need to work around the fact that C++ obfuscates what is actually going on when you call a method. You can do this by using a slightly obscure C++ construct called the *pointer to member*. For the present discussion the member in question will always be a member function, but keep in mind that pointers to members can work with data members as well. This use is probably even less common since if you have an object, or a pointer or reference to an object you can access public members directly as you would access members of a struct, or via an accessor function; I would think that pointers to data members are necessary only for serious runtime hacking or compiler implementation. But the key point here is that C++ pretends that both data members and member functions are part of classes, while CLOS and Dylan much more explicitly separate these concepts.

Pointers to member functions are uncommonly used, probably in part because of their somewhat obscure syntax. The only place I've seen them used, in fact, is the Darwin kernel's IOKit source code. Recently I've had a need to customize class behaviors at runtime without subclassing and without a complete refactoring of a class into modular pieces, so I've been investigating the construct again.

The syntax is awkward, but it is consistent with the general C school of thought which says that declarations are read backwards (right to left). In C, if you have an integer variable:

```
int i;
```

to make a pointer out of it, you insert an asterisk to the left of the variable name:

```
int * i;
```

Similarly, if you have a prototype for a function returning **int** and taking an **int** parameter:

```
int fn ( int );
```

to take a pointer to it, you insert an asterisk to the left of the name:

```
int * fn ( int );
```

but the order of evaluation means that the parser reads it as a function returning a pointer to **int**. So you have to change the order of evaluation using parentheses:

```
int ( *fn ) ( int );
```

Now **fn** means a pointer to a function, not a function returning a pointer.

A pointer to a *member function* is basically the same, with the addition of the class name and the scope resolution operator (::). When you are declaring your class, this prefix is not wanted or needed (it is implicit within the class declaration's curly braces), but when you define your methods, you do so like this:

```
int my_class::my_member_func( int )
{
   /* function body */
}
```

An aside: the separation between the class declaration and class definition is just one of the speed bumps you have to live with when using C++; it is there to support separate compilation, but if you've ever made the switch from C++ to Java you know that giving up the need to keep header files and implementation files in perfect synchronization gives you an immediate productivity boost.

If we want to make a variable that can point to that member function, we put an asterisk to the left of the name. You might guess it would look something like this:

```
int my_class::*my_member_func_ptr( int );
```

but that doesn't work. Te precedence rules get confused by this (during parsing, it probably looks like a collision between a definition of a member function and a conventional function returning a pointer with no type attached, or some such nonsense), so again we have to use parentheses:

```
int ( my_class::*my_member_func_ptr )( int );
```

and this gives us a variable, **my_member_func_ptr**, which can hold a pointer to a member function with the signature "returns **int** and takes one **int** parameter." Note that the parameter list, as in a function prototype or member function declaration, only requires the types of the parameters and not the names, although you can supply names for documentation purposes if you want.

Another aside: it would be easy to ridicule this syntax, but ridiculing the syntax of C++ is kind of like shooting fish in a barrel. I've shown that the syntax is at least somewhat consistent with the other C++ syntactical forms. Because of that, this particular syntax doesn't really bother me; if I can't remember it off the top of my head, I can mentally follow this derivation path and reason it out. This isn't necessarily true of a lot of other pieces of C++ syntax which do bother me and which I'm quite happy to rant about; see my piece on the **struct**.

A pointer to member function does not have a standard implementation; the implementation (size, etc.) may differ between implementations; you should probably treat it as an opaque data structure and make no assumptions about what it actually looks like inside, unless you want to write very non-portable code. There's nothing in the standard that says that pointers to member functions have to be the same size, for example, and the standard specifically says they are *not* just specially typed pointers; they are not convertible to and from **void*** by casting, as other pointer types are, using either the old-style or new-style casts.

This is because pointer to member function is not a usable *pointer* by itself; instead it contains whatever extra information is needed to represent a specific member function for a given class, probably an index or offset of some sort. You can imagine that the compiler needs a mechanism like this for its own use, so that it can turn

```
object.member_function(...)
```

or

```
object_pointer->member_function(...)
```

into C-style function calls with the addition of a hidden parameter:

```
looked_up_member_function( this, ... )
```

Remember our discussion on generic function dispatch? You might notice that when you rewrite the code to explicitly pass in the contest (the "this"), it starts to look like Dylan (or, ignoring the difference between infix and prefix notation, CLOS). You can actually think of the C++ object model as a much more limited and *obfuscated* form of CLOS generic function dispatch. How weird is that?

The "this" pointer allows the function body access to the specific object instance variables, and can also be passed on to any subsidiary member function calls on the same object. In Dylan or CLOS you would access the data members (in slots) of the incoming object *explicitly*; C++ gives you access to the data members and member functions of "this" implicitly, although you can still use **this->** as a prefix if you want to make very clear what you are doing, or protect against accidentally accessing something global or local in the member function's namespace that might inadvertently shadow a member. This merging of namespaces is another rich source of potential errors (remember my comment about how C++ is like *obfuscated* Dylan?)

Note that our pointer to member function is not defined to refer to *a specific* member function in the given class. It can have assigned to it *any* member function that is part of the class (or a derived class), and that has the same *signature* (that is, matching return type and parameter types). This, as you might imagine, allows the compiler to do dynamic dispatch based on runtime type; you can do the same thing, based on any criteria you choose.

Note also that the variable you've created is *not yet initialized*. There's that obfuscation again! In a language where everything is done by reference, that

isn't even quite possible; your reference will wind up referring to *something*. But pointers enable whole new classes of errors, and there is unfortunately no such thing as a "reference to member function" in C++. Be careful to make certain you initialize it before use, preferably at the point the variable is defined! Not doing so will cause you severe tire damage!

And, in fact, the compiler may not help you avoid this. I tested GCC by writing a call to an uninitialized regular function pointer:

```
void ( *fptr ) ( void );
fptr();
```

and it generated a helpful warning for this case. When I wrote a similar call using an uninitialized pointer to a member function, like so:

```
void ( Class_c::*mptr ) ( void );
( this->*mptr ) ();
```

GCC produced no warning at all, and as you might expect, the call caused an immediate crash. This is the likely outcome, but according to the C++ standard the behavior is *undefined* (this is very bad; it means the program is not required by the standard to catch this as an error; it is free to silently fail in some insidious way, or destroy your hard drive, or electrify your keyboard).

You can initialize them to zero, the *null pointer constant* (there is an explicitly allowed conversion), but the results of calling a null pointer to member function are *also* undefined, so this doesn't buy you anything; you are better off initializing them at the point where they are defined. Here is an example of how to do so:

```
void ( my_class::*mp ) ( void ) = &my_class::member_function;
```

Since pointers to member functions are hard to read, especially for member functions that that accept a long list of parameters, it is valuable to make the construct into a type using **typedef**. I recommend using the naming convention *class name + the _kind_ of member func + "_pm_t"*

But follow your own convention, or your team's convention, if you like. Keep in mind that the type can be used to represent *any* member function of the class (or a derived class) that matches the signature. The member function will probably represent a *handler* of some kind, or in design pattern terms a specific *strategy*. Describe what it handles, or what the strategies are trying to accomplish.

When you write the **typedef**, keep in mind that it looks just like the variable definition, except that the type name replaces the variable name and you put **typedef** in front. To define a type called **my_class_SomethingHandler_pm_t**, use the form:

```
typedef int ( my_class::*my_class_SomethingHandler_pm_t ) ( int );
```

Once you have a type, the following definition creates a variable with the new type and initializes it with a specific member function:

```
my_class_SomethingHandler_pm_t pm = &my_class::my_member_func;
```

This assignment can also take place in *parameter binding*. This means you can write a standalone function, member function, or static member function that accepts any pointer to member function matching the specified signature, and then pass it a specific one at run-time.

Once you have a pointer to member function, you can call that member function, but to do this you need an *object* (the "this" that will be in effect for the duration of the call). If you have a *pointer* to the object (which could be "this"), use the syntax:

```
( object_ptr->*mp )( params );
```

If you have a local or global variable holding an object or a *reference* to an object, use this syntax:

```
( object.*mp ) ( params );
```

Note that the parentheses around the first part are mandatory, to help the parser. If your object pointer or pointer to member function is stored in a structure or you are accessing it via pointer you will probably have to introduce more parentheses to make sure the parts of the expression are evaluated in the right order. This kind of construct can get ugly fast, so be careful.

In his C++ FAQ Lite section on pointers to member functions available here Marshall Cline suggest that when making calls using pointers to member functions, you always define a calling macro like this:

```
#define CALL_MEMBER_FN(object,ptrToMember) ((object).*(ptrToMember))
```

and then do the call like this:

```
result = CALL_MEMBER_FN(PO,PM)( PL );
```

but I wouldn't necessarily recommend that; C-style macros are still quite evil, and can make it very hard to find out what the compiler is complaining about, or even what it actually compiled that you didn't intend.
