Subject: Re: overload init function in class/object ?
Posted by JD Smith on Tue, 06 May 2003 21:32:45 GMT
View Forum Message <> Reply to Message

On Tue, 06 May 2003 13:34:06 -0700, David Fanning wrote:

> paul wisehart (wisehart@runbox.com) writes:
>
>>> I'm not sure what you mean by it, but it sure doesn't mean "define
>>> the same thing twice, in two different ways".
>> In C++ thats exectly what it means :) Furthermore, thats what the term
>> "function overloading" means in general. But, I see know that I was
>> incorrect about IDL's interpretation of "function overloading"
>
> Really!? I can see now why we are *both* confused, because that sure as
> hell doesn't make any sense to me! :-)
>

In statically types languages such as C++, and other optionally-typed languages, there is the notion of a subroutine's "signature", i.e. its return type, number of arguments, and each argument's type, taken together. Using the signature, a specific method can be chosen from among a number of methods with the same name, but differing signatures, e.g.:

```
class Example {
  // same name, three different methods
 int sum (int a) { return a; }
 int sum (int a, int b) { return a + b; }
 int sum (int a, int b, int c) { return a + b + c; }
}
```

Since such languages must figure out which version of the "sum" method to call (i.e. "bind" the method) at compile time, this is a nice way of preserving the semantics of an operation (like "sum") without resorting to ugly names like sum_one, sum_two, and sum_three. Since IDL is not statically-typed (and why would we want it to be), it has (almost) no notion of signature overloading, nor does it need it. The only situation where it does permit it relates to the duality of procedures vs. functions, which can be thought of as a minimal "binary" signature (e.g., returns nothing, or returns something). You are allowed to use a single name for both a procedure and function method within a class, e.g.:

```
pro Obj::Foo,f
   self.foo=f
end
```

```
function Obj::Foo
  return,self.foo
end
```

and IDL will happily distinguish

```
obj->Foo
```

from

```
foo=obj->Foo()
```

However, since arguments are always optional by default, and type is a run-time property, signature overloading is easily reproduced, and made even more convenient with keywords. Another bonus: you probably won't find yourself in need of signature overloading very often, since IDL's lack of type enforcement allows you to pass many more compatible data types into the same piece of code. Whereas C++ would need:

```
int sum(int a, int b)
float sum(float a, float b)
float *sum(float *a, float *b)
char *sum(char *a, char *b)
...
```

IDL can use the exact same method to add any numeric, array, or string data type.

Note that overloading contrasts markedly from "overriding", which IDL does support, and relates to the familiar parent/child method of replacement or refinement typified in statements like:

```
if (self->Parent::Init(_EXTRA=e) ne 1) then return,0
```

JD
