Subject: Re: Declaration of variables in IDL
Posted by Craig Markwardt on Wed, 03 Oct 2001 21:43:46 GMT
View Forum Message <> Reply to Message

"R.G.S." <rgs1967@hotmail.com> writes:

> Hans de Boer <boer@kfih.azr.nl> wrote in message
> news:3BBB0EC8.1CCB5E11@kfih.azr.nl...
>> Hi there,
>>
>> I have been working with IDL for quite a while now, but never
>> encountered a way of fixing variable types in IDL. Especially when
>> adding new variables to old pieces of code, it can be very confusing if
>> a variable of that name was already in use. IDL simply changes the
>> variable type (including changes of array dimension) if required and
>> possible. If not possible, you are the lucky one to get run-time errors,
>> but if possible, the code seems to run quite nicely and the poor
>> programmer is left with apparently correct results.
>> I also have found that people new to IDL have generated extremely sneaky
>> bugs this way.
>> Does anybody know of making IDL performing an explicit variable check
>> during compilation ?
>>
>> Thanks
>>
>> Hans deB
>
> As answered, IDL allows type changing. Such is life.

I agree 100%. The advantage of IDL is that the type of the variable can change dynamically. The disadvantage is that the type of the variable can change dynamically.

My approach is two-fold. First, test any input variables to a procedure to ensure they are the correct type AND DIMENSION to the ones you were expecting. If these attributes aren't correct then there are two options. Either error out, or try to recast the data to the correct type and dimension.

So, for example, if I am expecting a parameter to be a long integer scalar, then I usually cast it myself doing this:

```
param = round( param(0) )
```

This does two things, it ensures that param is a scalar, by scooping off the first element, and it also ensures that PARAM is cast to a long integer. The same thing applies when casting to strings, as in param = strtrim(param(0), 2)

Then things can get even more sticky, because you also need to decide whether you are going to modify the caller's original argument, or make a copy for yourself. Modifying the caller's argument can confuse your caller sometimes. It's usually not polite. For example, I usually modify the above approach to make a copy. The original parameter is PARAM0, and the internal copy is called PARAM.

```
param = round( param0(0) )
```

Also it is crucial to document the expected type and dimension of each parameter. That way the user can know ahead of time what to pass. If they pass the wrong thing then the blame can be assigned to them rather than you :-)

***

The second thing is to follow type-safe practices within your procedures. I am sure other people can come up with some good ideas. Here are some of mine.

The output of TOTAL is always a floating point type. Beware of this if you operate on integers and expect TOTAL to output an integer!

Beware of constructing an output variable by doing OUT = FLTARR(N). The problem with this is, what if the user is using double precision? You have just ruined their precision! Better to do something like this: OUT = IN * 0, in which case the output will be the same type as the input.

It can get more subtle than this. For example, what if you want your output to be a floating point type, and not an integer. The danger of the above approach is if the user passes an integer into the input variable IN, then OUT will also be an integer, undesireably. So usually what I do is: "OUT = IN * 0." Note the extra decimal point, which forces OUT to be at least floating point, but if IN is double precision, then the "0." is safely upcasted to double precision as well.

Beware of using constants like "0.0D" or "0L" in your formulae, because they may end up up-casting your data to a higher data type. Maybe that's okay, but sometimes not. I find myself using code like this:

```
zero = in(0) * 0
one  = zero + 1
```

at the beginning of a procedure. That way I can use ZERO and ONE

safely in place of 0 and 1 without worry of upcasting IN.

IDL will often drop the final dimension of an array if it is 1. This
is often fine, but occasionally it will really put a wrench in the
gears. If you really expect an array to have a fixed set of
dimensions, then you must be sure to REFORM it. I even do that right
after creating arrays!

```
  x = fltarr(nx, ny, nz)  ;; No guarantee NXxNYxNZ if NZ = 1!!!
  x = reform(x, nx, ny, nz, /overwrite)
```

Okay, enough for now!

Good luck,
Craig

--
-- ----------
Craig B. Markwardt, Ph.D. EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
-- ----------