Subject: Re: are there any s/w eng tools for IDL
Posted by William Clodius on Tue, 25 Feb 1997 08:00:00 GMT
View Forum Message <> Reply to Message

Mitchell R Grunes wrote:
>
> <snip>
>
> What is hungarian notation? Something like reverse-polish?

At one time some of the more publicized Microsoft programming efforts relied on Hungarian notation to provide discipline to C coding, and the publicity associated with this made it quite well known in some circles, but this notation has never been uniformly adopted by Microsoft.

From the comp.lang.c FAQ

http://phantom.iweb.net:80/docs/C/faq/q17.8.html

"Hungarian Notation is a naming convention, invented by Charles Simonyi, which encodes things about a variable's type (and perhaps its intended use) in its name. It is well-loved in some circles and roundly castigated in others. Its chief advantage is that it makes a variable's type or intended use obvious from its name; its chief disadvantage is that type information is not necessarily a worthwhile thing to carry around in the name of a variable."

Such a convention might require that all integers start with the letters, i, j, k, m, n, all logical variables start with an l, all pointers start with a p, all structures start with an s, etc.

In languages that perform implicit type changes such an naming convention will let you know when such a type change is occuring. Because type changes, such as from a LONG to an INTEGER, from DOUBLE to FLOAT, can introduce errors, such a convention can help make such changes explicit.

However if you want to change an implementation, from say an INTEGER to a LONG, then you have to change the name everywhere. If you want to make subtle distinctions, between an INTEGER and a LONG, between scalars and arrays, etc. a large portion of the name is occupied by unpronounceable gibberish. I am not a fan of the Hungarian notation.

>
> For the most part, type checking and even arguement checking simply
> aren't an issue.
>

I would disagree with this blank statement. I have been burned too many times trying to maintain other's code that played too fast and loose with dynamic typing. I would say however that it is not just dynamic typing, but its interaction with some coding styles that is a problem. There is a mistaken impression that reusing a name for multiple purposes can result in a performance or space savings. As a result, some programmers use names inappropriately to mean multiple things. It is one thing to use the same name, say data, for byte data and the floating point data that results after the appropriate calibration has been applied, it is another to use the name, say array, arbitrarilly within a set of code to represent any possible array.

Note that while I can accept using the same name for related quantities, I would prefer however not to use the same name for calibrated and uncalibrated data. My preferred style might be

final_data = Calibrate(Temporary(raw_data), calibration_data)

if raw_data is not going to be used subsequently.

> <snip>
>
> There are several software engineering issues that do arise:
>
> 1. As mentioned above, when you operate on arguements of a
> function or procedure, you are also operating on the values in the
> calling program. C/C++ ordinarally make local copies of scalar
> values. This is one of the many ways in which IDL was designed to
> be Fortran-like, not C-like.

Try to avoid changes in the types of arguments as it typically represents a change in the meaning that is not obvious in the calling code. Try to avoid modifying arguments to functions, as opposed to procedures, as functions that modify their arguments is often counterintuitive and can result in subtle errors.

>
> 2. Integer arithmetic overflows and underflows are not detected.
> For example, 1024*1024 would yield 0 on most platforms, because
> small integers are stored in 16 bits, and 16 bit arithmetic on
> almost all modern computers is actually arithmetic modulo 2^16 (if
> you think about it, that is even true of two's complement signed
> integers). That "defect" is also true of most C compilers, by the
> way--in fact it is common C programming practice to take advantage
> of that. (Some Fortran compilers have a switch which lets them
> detect that sort of error--a good advantage of Fortran :-).

Note also that by default IDL integers are 16 bit not 32 bit. This can result in some subtle errors. It is good practice to make every integer constant a long by appending an L, (and hence making the associated varibale a long) unless you are certain that only 16 bits are necessary.

> <snip>
>
> 4. Argument checking is mostly not a problem, since it mostly just
> leads to run-time errors when you try to use them. If your calling
> program has more arguments than the called program, or includes
> keyword arguments that the called program is missing, there will be
> a run time error. If the reverse is true, there will be no error.
> However, the unspecified variables will be undefined--that is
> n_elements(variable)
> will be 0--see variable d earlier in this post.

Mostly but not always not a problem. I have been burned when say a program assumes that an argument is a long and say has
a = a + 1L
that converts an integer array to a long array and exceeds memory limits, (very rare)

or an input that is a byte array where something like

a = a^2

causes an undetected overflow for BYTE arrays that would not occur for
a = LONG(TEMPORARY(a))^2

Note these and other errors are contest dependent, and are best addressed by good documentation not naming conventions.

> <snip>

--

William B. Clodius Phone: (505)-665-9370
Los Alamos Nat. Lab., NIS-2 FAX: (505)-667-3815
PO Box 1663, MS-C323 Group office: (505)-667-5776
Los Alamos, NM 87545 Email: wclodius@lanl.gov