Subject: compile a routine wich inlude a commun
Posted by L. Testut on Sun, 22 Jan 2006 12:26:58 GMT
View Forum Message <> Reply to Message

Dear IDL users,
I am a beginner with IDL and I really enjoy this programmation langage, even if it is not easy at the beginning. I'm building at the moment an application to work with satellite altimetric data of four satellites.
My question is "it is possible to compile a procedure or function including a COMMON which is not already defined" in others words : "is it possible to force compilation of procedure or function including a COMMON which is not already defined" ?

Your answer will probably be : don't use common at all ! (but I don't know how to do without common)
Regards,
Laurent Testut
Toulouse, France

Subject: Re: compile a routine wich inlude a commun
Posted by David Fanning on Mon, 23 Jan 2006 18:04:39 GMT
View Forum Message <> Reply to Message

David Fanning writes:

> Oh, well, I mean "exactly in the IDL sort of way".
> You know what I mean. Kind of in the way NLEVELS
> gives you exactly that many levels in a Contour plot. :-)

By the way, I've been meaning all morning to point
out that a sense of humor and a taste for ambiguity
and chaos goes a long way toward making IDL programming
more palatable. Control freaks should probably look
elsewhere. :-)

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/

Subject: Re: compile a routine wich inlude a commun
Posted by Paul Van Delst[1] on Mon, 23 Jan 2006 18:25:10 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
> David Fanning writes:
>
>
>> Oh, well, I mean "exactly in the IDL sort of way".
>> You know what I mean. Kind of in the way NLEVELS
>> gives you exactly that many levels in a Contour plot. :-)
>
>
> By the way, I've been meaning all morning to point
> out that a sense of humor and a taste for ambiguity
> and chaos goes a long way toward making IDL programming
> more palatable. Control freaks should probably look
> elsewhere. :-)

So, what're you saying? IDL is non-deterministic?

Wow. That's pretty impressive. Not good for a programming language, but still impressive. :o)

paulv

--
Paul van Delst
CIMSS @ NOAA/NCEP/EMC

Subject: Re: compile a routine wich inlude a commun
Posted by David Fanning on Mon, 23 Jan 2006 18:38:52 GMT
View Forum Message <> Reply to Message

Paul Van Delst writes:

> So, what're you saying? IDL is non-deterministic?
> Wow. That's pretty impressive. Not good for a programming language, but still impressive. :o)

That's all I'm saying... :-)

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.

Subject: Re: compile a routine wich inlude a commun
Posted by David Fanning on Mon, 23 Jan 2006 23:16:15 GMT
View Forum Message <> Reply to Message

Paul Van Delst writes:

> Well, they're called pointers but they're not, really. You can't actually "point" to
> anything - just make copies.

Well, they aren't fingers, if that's what you mean. :-)

> But, not being a pointer expert, let me ask the question:
> How *do* you use a pointer in IDL to, uh, well, "point" to an already created variable? Or
> just parts of an already created array?

They are variables that live in a different place
than local variables. Outside the room, if you like.
A pointer can't point to the memory locations of your local
variable because there aren't enough technical support
engineers in the world to sort out why your programs
are suddenly crashing right and left.

Goodness, *all* variables work like this, as far as I
can tell.

```
 a = 5
 b = a
 b = 3
 print, a
```

And I don't think you really want to see a 3 printed out!
Chaos (and I have a high tolerance, myself) would reign supreme.

But, there is no reason you can do what you want to do.

```
  ; Create a local variable.
  x = Indgen(4,4)

  ; Move the variable out of the room.
  ptr = Ptr_New(x, /No_Copy)
  help, x
    X   UNDEFINED = <Undefined>

  ; Change a subset of the data
  (*ptr)[1:2, 1:2] = (*ptr)[1:2, 1:2] * 100
```

```
   ; Move the variable back into the room
   x = Temporary(*ptr)
   Help, *ptr
     <PtrHeapVar2>   UNDEFINED = <Undefined>

   Print, x
       0       1       2       3
       4     500     600       7
       8     900    1000      11
      12      13      14      15
```

The advantage of pointers as variables, is that they don't
go away when you exit the local program unit, so they can
be used by any program unit made aware of them. That's
pretty powerful.

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/

Subject: Re: compile a routine wich inlude a commun
Posted by Paul Van Delst[1] on Tue, 24 Jan 2006 00:14:08 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
> Paul Van Delst writes:
>
>
>> Well, they're called pointers but they're not, really. You can't actually "point" to
>> anything - just make copies.
>
>
> Well, they aren't fingers, if that's what you mean. :-)

Actually, yes, that's pretty much exactly what I mean. Hence the name: pointers.

>> But, not being a pointer expert, let me ask the question:
>> How *do* you use a pointer in IDL to, uh, well, "point" to an already created variable? Or
>> just parts of an already created array?
>
>

> They are variables that live in a different place
> than local variables.

Well, then they're *not* pointers. They're, umm, "copy/move enablers". :o)

> Outside the room, if you like.
> A pointer can't point to the memory locations of your local
> variable because there aren't enough technical support
> engineers in the world to sort out why your programs
> are suddenly crashing right and left.
>
> Goodness, *all* variables work like this, as far as I
> can tell.
>
```
>   a = 5
>   b = a
>   b = 3
>   print, a
```
>
> And I don't think you really want to see a 3 printed out!

Sorry, but I don't see the analogy here at all.

```
  a=5
  b=>a (b points to a)
  a=3
  print, *b
```

should give you "3", and

```
  *b=7
  print, a
```

should give you "7". Because in this example b is a (actual) pointer, it refers to the
same memory as the variable a. Changing either via assignment changes the value in the
memory location.

> Chaos (and I have a high tolerance, myself) would reign supreme.

Erm, not really. This is how actual pointers work (pretty much).

The IDL analogy of a "pointer" is

```
   a=5
   b=a
   b=3
   a=b
   print, a
```

i.e. it's just copying stuff. Not pointing.

```
>
>  But, there is no reason you can do what you want to do.
>
>     ; Create a local variable.
>     x = Indgen(4,4)
>
>     ; Move the variable out of the room.
>     ptr = Ptr_New(x, /No_Copy)
>     help, x
>        X   UNDEFINED = <Undefined>
```

See, here's the problem. You can't point to a named variable with IDL pointers. You can copy it onto the heap, but not point to it.

```
>
>     ; Change a subset of the data
>     (*ptr)[1:2, 1:2] = (*ptr)[1:2, 1:2] * 100
>
>     ; Move the variable back into the room
>     x = Temporary(*ptr)
>     Help, *ptr
>       <PtrHeapVar2>   UNDEFINED = <Undefined>
>
>     Print, x
>        0       1       2       3
>        4     500     600       7
>        8     900    1000      11
>       12      13      14      15
```

I take issue with the need to "move" these variables around at all. They're pointers. You shouldn't have to move anything. That's, well, the point.

How is the above example any different from

```
IDL> x = Indgen(4,4)
IDL> ptr=temporary(x)
IDL> help, x
X               UNDEFINED = <Undefined>
IDL> ptr[1:2,1:2]=ptr[1:2,1:2]*100
IDL> x=temporary(ptr)
IDL> print, x
       0       1       2       3
       4     500     600       7
       8     900    1000      11
      12      13      14      15
```

??

In both cases you're effectively copying data

I use pointers in Fortran95 for only two things, different things.
1) Point to some nameless space via an allocate statement, e.g.
```
   integer,pointer::x(:)
   allocate(x(100))
```
This is similar to the IDL statement
```
   x = PTR_NEW(lonarr(100))
```
2) Alias subsections of an array, or different arrays
```
   integer,target::a(100,100),b(30)
   integer,pointer::x(:)
   ! Make x point to b. Modify elements of x and b changes
   x=>b
   ! Now make x point to slice of a. Modify elements of x and a(:,20) changes
   x=>a(:,20)
```

As far as I can tell, there is no equivalent to the above (2) in IDL. The ability to create an alias for a data object without having to copy it - *that's* powerful.

I must be missing something, somewhere.

Anyway, I gotta get home before the bottle shop shuts! :o)

cheers,

paulv

--
Paul van Delst
CIMSS @ NOAA/NCEP/EMC

Subject: Re: compile a routine wich inlude a commun
Posted by David Fanning on Tue, 24 Jan 2006 04:03:27 GMT
View Forum Message <> Reply to Message

Paul Van Delst writes:

> Well, then they're *not* pointers. They're, umm, "copy/move enablers". :o)

I have a feeling JD is going to come to our rescue
and sort the whole thing out for us. At least, I
hope so. :-)

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/

Subject: Re: compile a routine wich inlude a commun
Posted by peter.albert@gmx.de on Tue, 24 Jan 2006 08:09:01 GMT
View Forum Message <> Reply to Message

Just my 2 cents: imho, the problem with the given example

p=ptr_new(x[1:2,1:2])

is that x[1:2, 1:2] on the right hand side of the equation is, if I
remember correctly, actually a new temporary variable. So p is pointing
to a temporary variable which has nothing in commion with the variable
x, apart from the fact that its initial values equal the appropriate
ones of x. After that, any operation on p is totally disconnected to x.
On the other hand, Davids example used

p = ptr_new(x)

and here, p is actually pointing to the memory space occupied by x, so
modifying *p actually does modify x.

And yes, for this little example the effect is the same when using
temporary() two times, as Paul suggested. But this is not the, umh,
only point of pointers. I can hardly imagine how something like Davids
highly appreciated linked list object would work using temporary()
instead of pointers.

But well, looking at Pauls original wish of using pointers to alias
subsets of an array; given the fact that x[...] actually creates a new
temporary variable makes me feel that this is actually not possible in
IDL. Of course, there might be a way using histogram ...

Cheers,

Peter

Subject: Re: compile a routine wich inlude a commun
Posted by Maarten[1] on Tue, 24 Jan 2006 10:04:01 GMT
View Forum Message <> Reply to Message

JD Smith wrote:

> I think he means pointers are a kludge for extensible arrays.

I think you can read minds, that is what I meant.

> *But*, and this is a big but, all that flexibility comes at
> some real cost in speed, which grows with the data size, perhaps
> non-linearly.

Understood, and at times: accepted.

[snip, reasons to use IDL, and if speed is added, the "flexible"
languages may turn into a two-lobed "mess" as well]

> 6) Want to share code which just runs with colleagues, avoiding the
> package dependency and moving target problems of roll your own
> solutions like Python + numarray, or numeric, or numpy, ...
>
> Of course, this should include a footnote of {Rich colleagues who can
> afford IDL licenses}.

I can see that, and it is one of the reasons I haven't switched to
Python yet. The footnote adds one of my private adversions: at work I
can use IDL, good ideas occuring at home may go wasted.

[snip: links to IDL <---> python. thanks]

> Another thing you'll notice with most of these packages (and, sadly,
> even GDL) -- plotting is typically a compromise, borrowing a
> pre-existing package like GnuPlot, or matplotlib, not very cleanly
> integrated. It's a real pain to integrate decent graphics in a
> compatible, cross-platform way. I think this problem will eventually
> be solved, but for now, if I send you a Python+numpy+matplotlib
> script, it probably wouldn't run out of the box.

Agreed. Cairo Graphics (http://cairographics.org/introduction) seem
promising, but "not there yet". For my thesis I didn't like the output
of any of the graphics packages, so the final output was created in
MetaPost. Today I might have choosen PyX.

[on the Drizzling page at David's site]

> As one of the perpetrators of that page, I have to agree, these
> examples (and many of the IDL Way) are not terribly obvious. Some
> have maintenance concerns, to be sure. But, they enable you, in a
> typeless language, to obtain the kind of speed of operation on large
> (many MB to many GB) piles of data that are simply otherwise unheard

> of.

I think we disagree where the balance between readability and execution
speed lies.

> Moreover, a elegant Python Drizzle would probably run 10x slower even
> than the straightforward loop-based IDL drizzle. At some point, you
> give up and write it quite simply in C, spending 95% of the time and C
> code figuring out how to communicate the results back with IDL. So, I
> agree with the original poster that the algorithms mentioned, among
> many others in IDL, are not at all transparent, while simple versions
> of the same are not at all fast. However, in my experience, this is
> the price you pay in the tradeoff of elegance and raw speed.

I prefer my "normal" everyday use to be elegant, and when I really need
the speed (after profiling, no need for premature optimization), an
easy route to C, Fortran, (assembly, no, I'm not that nuts) is
appreciated.

> I think if RSI wants to do one thing to move the tradeoff forward,
> they should take MAKE_DLL and vastly expand its scope, allowing you to
> trivially stick *simple* bits of C-code callout which operate directly
> on IDL data in memory.

I have some experience with inserting C code into Labview. There are
several methods to do so: one truly integrated, and I think it is about
as hard to do as in IDL, and probably harder to debug. Another method
is to call into a shared library. As long as you leave the memory
allocation of anything that gets communicated back into (Labview/IDL)
to the calling application, it is near trivial - just be careful to
clean up after yourself.

It wasn't at the level where pyrex lives, but easy enough, and a
compilable shell for debugging was easy enough to generate.

I hope IDL will get to that point - or maybe I've not found the right
sample code to do this (especially on a 2D array). A pity that SWIG
doesn't support IDL, it might make some things quite a bit easier.

> Python has several projects aiming to do just
> this, and if any of them become standard, this may change the
> scientific coding landscape significantly.

I think that is a given, it is only a matter of time. Another project
to keep an eye on is PyTables: transparent persistent storage of arrays
in an HDF file with a decent speed.

Maarten

Subject: Re: compile a routine wich inlude a commun
Posted by Paul Van Delst[1] on Tue, 24 Jan 2006 14:42:18 GMT
View Forum Message <> Reply to Message

Peter Albert wrote:
> Just my 2 cents: imho, the problem with the given example
>
> p=ptr_new(x[1:2,1:2])
>
> is that x[1:2, 1:2] on the right hand side of the equation is, if I
> remember correctly, actually a new temporary variable. So p is pointing
> to a temporary variable which has nothing in commion with the variable
> x, apart from the fact that its initial values equal the appropriate
> ones of x. After that, any operation on p is totally disconnected to x.
> On the other hand, Davids example used
>
> p = ptr_new(x)
>
> and here, p is actually pointing to the memory space occupied by x, so
> modifying *p actually does modify x.

No, it does not. Whether or not you point to the "complete" variable, or just a part of it, a new heap "variable" is created.

```
IDL> x=indgen(4,4)
IDL> print, x
       0       1       2       3
       4       5       6       7
       8       9      10      11
      12      13      14      15
IDL> p=ptr_new(x)
IDL> print, *p
       0       1       2       3
       4       5       6       7
       8       9      10      11
      12      13      14      15
IDL> print, (*p)[1:2,1:2]
       5       6
       9      10
IDL> (*p)[1:2,1:2] = (*p)[1:2,1:2] + 100
IDL> print, *p
       0       1       2       3
       4     105     106       7
       8     109     110      11
      12      13      14      15
IDL> print, x
       0       1       2       3
       4       5       6       7
       8       9      10      11
      12      13      14      15
```

> And yes, for this little example the effect is the same when using
> temporary() two times, as Paul suggested. But this is not the, umh,
> only point of pointers. I can hardly imagine how something like Davids
> highly appreciated linked list object would work using temporary()
> instead of pointers.

I agree, but you create a link list object once and that's pretty much it. If you've done your job right, you don't have to do much else except use the object. On the other hand, aliasing is something that comes in handy in day-to-day usage. My Fortran95 linked list code hasn't been touched pretty much since I wrote it. But I use aliasing a lot to, for example, concatenate complicated and large data structures. Aliasing let's me manipulate the structures more naturally with minimum copying/creation of temporary vars.

> But well, looking at Pauls original wish of using pointers to alias
> subsets of an array; given the fact that x[...] actually creates a new
> temporary variable makes me feel that this is actually not possible in
> IDL.

That's what I'm thinking also. There's probably some fundamental design issue that prevents it from being a simple thing to do. Sort of like passing structure components as arguments and expecting them to be writable in the called procedure - the pass-by-reference/value issue previosuly mentioned.

> Of course, there might be a way using histogram ...

That's possible :o) But, then, they would be histopointers. Or maybe pointygrams? :o)

paulv

--
Paul van Delst
CIMSS @ NOAA/NCEP/EMC

Subject: Re: compile a routine wich inlude a commun
Posted by peter.albert@gmx.de on Tue, 24 Jan 2006 16:01:18 GMT

View Forum Message <> Reply to Message

```
> No, it does not. Whether or not you point to the "complete" variable, or just a part of
> it, a new heap "variable" is created.
>
> IDL> x=indgen(4,4)
> IDL> print, x
>       0      1      2      3
>       4      5      6      7
>       8      9     10     11
>       12     13     14     15
>  IDL> p=ptr_new(x)
>  IDL> print, *p
>        0      1      2      3
>        4      5      6      7
>        8      9     10     11
>       12     13     14     15
>  IDL> print, (*p)[1:2,1:2]
>        5      6
>        9     10
>  IDL> (*p)[1:2,1:2] = (*p)[1:2,1:2] + 100
>  IDL> print, *p
>        0      1      2      3
>        4    105    106      7
>        8    109    110     11
>       12     13     14     15
>  IDL> print, x
>        0      1      2      3
>        4      5      6      7
>        8      9     10     11
>       12     13     14     15
```

Damn, I should have tried it before. I was so sure :-(

But then, this _is_ a polite group. Nobody murmuring something like
rtfm, while a look in the manual reveals :

"If InitExpr is provided, PTR_NEW uses it to initialize the newly
created heap variable. Note that the new heap variable does not point
at the InitExpr variable in any sense-the new heap variable simply
contains a copy of its value."

Well, here is is, black on white. So you _can't_ alias variables or
subarrays.

Cheers,

Peter

Subject: Re: compile a routine wich inlude a commun
Posted by Paul Van Delst[1] on Tue, 24 Jan 2006 16:09:56 GMT

View Forum Message <> Reply to Message

Peter Albert wrote:
>> No, it does not. Whether or not you point to the "complete" variable, or just a part of
>> it, a new heap "variable" is created.
>>

```
>> IDL> x=indgen(4,4)
>> IDL> print, x
>>        0       1       2       3
>>        4       5       6       7
>>        8       9      10      11
>>       12      13      14      15
>> IDL> p=ptr_new(x)
>> IDL> print, *p
>>        0       1       2       3
>>        4       5       6       7
>>        8       9      10      11
>>       12      13      14      15
>> IDL> print, (*p)[1:2,1:2]
>>        5       6
>>        9      10
>> IDL> (*p)[1:2,1:2] = (*p)[1:2,1:2] + 100
>> IDL> print, *p
>>        0       1       2       3
>>        4     105     106       7
>>        8     109     110      11
>>       12      13      14      15
>> IDL> print, x
>>        0       1       2       3
>>        4       5       6       7
>>        8       9      10      11
>>       12      13      14      15
```
>
>
> Damn, I should have tried it before. I was so sure :-(
>
> But then, this _is_ a polite group. Nobody murmuring something like
> rtfm, while a look in the manual reveals :
>
> "If InitExpr is provided, PTR_NEW uses it to initialize the newly
> created heap variable. Note that the new heap variable does not point
> at the InitExpr variable in any sense-the new heap variable simply
> contains a copy of its value."
>
> Well, here is is, black on white. So you _can't_ alias variables or
> subarrays.

Huh, how 'bout that? It's in the manual. (ehem)

I wonder why RSI decided to implement pointers this way? Or, to rephrase, why they called the implementation they chose a pointer?

paulv

--
Paul van Delst
CIMSS @ NOAA/NCEP/EMC

Subject: Re: compile a routine wich inlude a commun
Posted by Paul Van Delst[1] on Tue, 24 Jan 2006 16:13:02 GMT
View Forum Message <> Reply to Message

Paul Van Delst wrote:
> Huh, how 'bout that? It's in the manual. (ehem)

Just to be painfully clear, the above is not sarcasm. I was actually surprised to see that the manual states it that clearly. I should've read the online help earlier and preserved some electrons.

paulv

--
Paul van Delst
CIMSS @ NOAA/NCEP/EMC

Subject: Re: compile a routine wich inlude a commun
Posted by David Fanning on Tue, 24 Jan 2006 16:26:44 GMT
View Forum Message <> Reply to Message

Paul Van Delst writes:

> I wonder why RSI decided to implement pointers this way? Or, to rephrase, why they called
> the implementation they chose a pointer?

Originally they were called "handles", but that annoyed
pretty much everyone. :-)

Cheers,

David

P.S. And before *that* they were called user values of
unrealized top-level base widgets. But that's really going
too far back in history, probably. :-)

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.dfanning.com/

Subject: Re: compile a routine wich inlude a commun
Posted by Paul Van Delst[1] on Tue, 24 Jan 2006 16:58:20 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
> Paul Van Delst writes:
>
>
>> I wonder why RSI decided to implement pointers this way? Or, to rephrase, why they called
>> the implementation they chose a pointer?
>
>
> Originally they were called "handles", but that annoyed
> pretty much everyone. :-)

Ah, yes. Good old handles. How did we widget-program without them? I guess I shouldn't be too hard on the IDL naming conventions. Coming up with a descriptive, abstract name for something can be difficult.

paulv

>
> Cheers,
>
> David
>
> P.S. And before *that* they were called user values of
> unrealized top-level base widgets. But that's really going
> too far back in history, probably. :-)
>

--
Paul van Delst
CIMSS @ NOAA/NCEP/EMC

Subject: Re: compile a routine wich inlude a commun
Posted by JD Smith on Tue, 24 Jan 2006 20:55:35 GMT
View Forum Message <> Reply to Message

On Mon, 23 Jan 2006 21:03:27 -0700, David Fanning wrote:

> Paul Van Delst writes:
>
>> Well, then they're *not* pointers. They're, umm, "copy/move enablers".
>> :o)
>

> I have a feeling JD is going to come to our rescue and sort the whole
> thing out for us. At least, I *hope* so. :-)

IDL pointers are really properly called "references", and are similar to references in many other languages, like Perl. In some ways, they are more limited, since they can only reference a special pool of variables which are otherwise inaccessible: the "heap variables".

In many languages, there are two ways to make a reference: a reference can be made of a pre-existing normal variable, or a new "anonymous" reference can be made, essentially referring to a freshly made heap variable. IDL only supports on the latter method -- anonymous references -- not the former method. There is no "address-of" operator, so there is no way to take an address of an existing variable, ala:

```
IDL> x=2
IDL> new_ptr=&x ;; No such thing
```

It just doesn't exist. Pointer heap variables and normal variables will forever live their separate lives. You *can* cheaply re-assign the actual memory pointed to by a regular variable to a pointer heap variable, and visa versa, using the method David already outlined:

```
IDL> x=indgen(1000000L)
IDL> p=ptr_new(x,/NO_COPY) ;x now undefined
IDL> x=temporary(*p) ;*p now undefined
```

But at no time can you have more than one reference to a given normal IDL variable (e.g. two ways to modify it). You can, of course, have multiple references to each pointer heap variable (or none at all, which is a great way to leak memory). This is covered in the pointer tutorial.

One very basic fact about IDL pointers is that they are very heavy, both in terms of compute time, and memory. Try this:

```
IDL> m=memory(/current)
IDL> a=ptrarr(10000L,/ALLOCATE_HEAP) & for i=0,10000L-1 do *a[i]=1L
IDL> print,(memory(/current)-m)/1024
        382
IDL> a=0
IDL> m=memory(/current)
IDL> a=replicate(1L,10000L)
IDL> print,(memory(/current)-m)/1024
         39
```

Using a pointer to store 10000 long integers (1L) takes approximately

10x as much memory as storing those 10000 integers directly in an array. This is because even *empty* IDL variables take up around 35 bytes of memory. All this extra memory goes to all the additional information associated with each and every IDL variable (mostly things like variable type, array lengths, etc. -- see idl_export.h if you are curious).

Here's how to see that:

```
IDL> a=0
IDL> m=memory(/current)
IDL> a=ptrarr(10000L,/ALLOCATE_HEAP)
IDL> print,float(memory(/current)-m)/10000L
     36.2015
```

So, "pointer" is really a bad name; "anonymous reference" would be a better (if longer) name. The problem with "pointer" is it is comes with no small amount of baggage from association with pointers in languages like C. Unlike C pointers, which are very lightweight, these "pointers" can't really be used for large, nested data structures, without a fairly large memory and speed penalty (not to mention the penalty that looping over many pointers entails). For this reason, the best use of pointers in IDL is to provide persistent, flexible storage for more typical IDL variable types, like large arrays or structures, which can be accessed and operated on without this penalty. Also unlike C pointers, IDL pointers can't be used to access or write memory outside of the IDL address space, so in that sense are much safer than C pointers.

JD

Subject: Re: compile a routine wich inlude a commun
Posted by JD Smith on Tue, 24 Jan 2006 21:05:09 GMT
View Forum Message <> Reply to Message

>
> That's what I'm thinking also. There's probably some fundamental design
> issue that prevents it from being a simple thing to do. Sort of like
> passing structure components as arguments and expecting them to be
> writable in the called procedure - the pass-by-reference/value issue
> previosuly mentioned.

The design issue is that since normal variables can't reference subsets of arrays of other variables, neither can pointer variables, since they are exactly the same thing.

By the way, IDL always does pass all arguments by reference, it just

happens that if you feed a routine a temporary variable, like a
structure member, or array subset, that variable is freed after the
call... so it only *seems* these are being passed by value. In fact,
modifying such arguments modifies the temporary variable, which is of
course, immediately deleted, and so was quite a useless operation.

JD

Subject: Re: compile a routine wich inlude a commun
Posted by JD Smith on Tue, 24 Jan 2006 21:13:23 GMT

View Forum Message <> Reply to Message

On Tue, 24 Jan 2006 02:04:01 -0800, Maarten wrote:

> JD Smith wrote:
>
>> I think he means pointers are a kludge for extensible arrays.
>
> I think you can read minds, that is what I meant.

Not mind reading, just shared annoyance.

>> As one of the perpetrators of that page, I have to agree, these
>> examples (and many of the IDL Way) are not terribly obvious. Some
>> have maintenance concerns, to be sure. But, they enable you, in a
>> typeless language, to obtain the kind of speed of operation on large
>> (many MB to many GB) piles of data that are simply otherwise unheard
>> of.
>
> I think we disagree where the balance between readability and execution
> speed lies.

The balance, I find, depends quite sensitively on the reader.

>> Moreover, a elegant Python Drizzle would probably run 10x slower even
>> than the straightforward loop-based IDL drizzle. At some point, you
>> give up and write it quite simply in C, spending 95% of the time and C
>> code figuring out how to communicate the results back with IDL. So, I
>> agree with the original poster that the algorithms mentioned, among
>> many others in IDL, are not at all transparent, while simple versions
>> of the same are not at all fast. However, in my experience, this is
>> the price you pay in the tradeoff of elegance and raw speed.
>
> I prefer my "normal" everyday use to be elegant, and when I really need
> the speed (after profiling, no need for premature optimization), an
> easy route to C, Fortran, (assembly, no, I'm not that nuts) is
> appreciated.

Elegance is also fairly subjective, especially when it comes to IDL
programming. After you immerse yourself in vector-based programming for
long enough, the sight of a loop is highly unsettling, not just for the
implicit speed penalty it may well entail, but in terms of perceived
elegance as well. Whether anyone else in the world would share that
aesthetic is debatable.

JD

Subject: Re: compile a routine wich inlude a commun
Posted by Foldy Lajos on Tue, 24 Jan 2006 21:30:22 GMT
View Forum Message <> Reply to Message

Hi,

and the odd thing is, that functions sometimes return these references,
not the values:

```
function f, arg
return, arg
end

a=1
(f(a))=2
help, a

A               INT       =        2
```

regards,
lajos

On Tue, 24 Jan 2006, JD Smith wrote:

> By the way, IDL always does pass all arguments by reference, it just
> happens that if you feed a routine a temporary variable, like a
> structure member, or array subset, that variable is freed after the
> call... so it only *seems* these are being passed by value. In fact,
> modifying such arguments modifies the temporary variable, which is of
> course, immediately deleted, and so was quite a useless operation.
>
> JD
>
>

Subject: Re: compile a routine wich inlude a commun
Posted by JD Smith on Tue, 24 Jan 2006 22:15:36 GMT
View Forum Message <> Reply to Message

On Tue, 24 Jan 2006 22:30:22 +0100, Fï¿½LDY Lajos wrote:

```
> Hi,
>
> and the odd thing is, that functions sometimes return these references,
> not the values:
>
> function f, arg
> return, arg
> end
>
> a=1
> (f(a))=2
> help, a
>
> A               INT       =        2
>
```

Interesting. Try:

```
IDL> (f(a[0]))=2
% Expression must be named variable in this context: <INT      (       2)>.
% Execution halted at: $MAIN$
```

and you'll see the distinction. Temporary variables can be used for many things, but direct assignment is not possible. One good use of temporary variables is subscripting or structure dereferencing them:

```
IDL> a=(size(b,/DIMENSIONS)[0]
```

JD