Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by pgrigis on Mon, 16 May 2011 20:46:18 GMT
View Forum Message <> Reply to Message

On May 16, 1:27 pm, JDS <jdtsmith.nos...@yahoo.com> wrote:
> A few years back, we had long discussions about the trouble the new "dot" syntax for method invocation would bring about. I've just stumbled across a new and unexpected case. Not only does IDL 8 interpret "." as equivalent to "->", in certain uses it also goes the OTHER WAY, recasting "->" as "." and overriding a method invocation with structure variable dereferencing, *even when the arrow operator is used*. This is a major issue for any pre-existing class which contains a method-function and structure member with identical names, which is quite common.
>
> Consider this simple class:
>
```
>  pro DOT_ARROW::try
>    ;compile_opt idl2
>    print,"GOT: ",self->item([1,2,3,4])
>  end
>
>  function DOT_ARROW::item, p
>    return, size(p,/DIMENSIONS)
>  end
>
>  pro dot_arrow__define
>    st={DOT_ARROW,$
>        item:0.0}
>  end
>
>  IDL> d=obj_new('dot_arrow')
>  IDL> d->try
>  GOT:       0.00000      0.00000      0.00000      0.00000   ;; RUNS without
ERROR, but is WRONG!!!
```
>
> Now with IDL2 enabled (or in IDL 7):
>
```
>  IDL> d->try
>  GOT:            4  ;; RIGHT!!!
```
>
> IDL 8 has gone one step further than introducing a new ambiguity between "." and "->" in the name of Python/JS-esque cosmetics, it's completed that ambiguity -- essentially enforcing it on us -- by making it work both ways. In essence, IDL8 is interpreting:
>
```
>    self->item([1,2,3,4])
```
>
> as
>
```
>    self.item[[1,2,3,4]]
```
>

> (no indexing error since you can't go out of bounds with an indexing vector by default). If I say "->", I mean invoke a method, no matter what. However, as new programmers adopt "." for method invocation, this ambiguity will never go away.
>
> And this is not a simple matter of troublesome syntax errors. What's really scary about this is, there are plenty of imaginable cases in which a->b(c) is interpreted by IDL8 in a completely different way from earlier versions of IDL, yet it can run through without error, silently corrupting your output. There is no warning against or detection of the use of identical names for a method-function and a class structure variable.
>
> Chris Torrence noticed this ambiguity could cause trouble while IDL 8 was being designed; I don't recall what the final decision was. But in any case, my understanding was this would be a new ambiguity, affecting new code which uses the dot operator for method invocation. Never was it discussed that IDL 8 would render inoperable (or worse, operable but incorrect) existing, functioning code by re-defining dereferencing post facto. I can scarcely see how this marketing-driven syntax change was worth it.
>
> JD

Wow! If anybody was still looking for reasons to delay upgrading,
that provides the strongest argument to date...

Ciao,
Paolo

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by Paul Van Delst[1] on Mon, 16 May 2011 21:03:36 GMT

View Forum Message <> Reply to Message

Crikey. I haven't yet verified any weird behaviour of the sort you describe in our IDL codes, but thanks for the heads up.

Your post has been saved to my desktop (in 12years, I've only done that 4 times) for immediate reference.

End-of-last year we went through a relatively big transformation of regular old functional code to an object library.
The code in question is the start of our processing chain so it's, you know, sorta important that that library work
correctly. And reliably so. In the background. The plausible-but-wrong answer is what keeps me up at night.

If things start going pear shaped, at least I have a prototype for conversion of said IDL object library to Fortran2003....

<Sigh>

cheers,

paulv

JDS wrote:
> A few years back, we had long discussions about the trouble the new "dot" syntax for method invocation would bring
> about. I've just stumbled across a new and unexpected case. Not only does IDL 8 interpret "." as equivalent to
> "->", in certain uses it also goes the OTHER WAY, recasting "->" as "." and overriding a method invocation with
> structure variable dereferencing, *even when the arrow operator is used*. This is a major issue for any pre-existing
> class which contains a method-function and structure member with identical names, which is quite common.
>
> Consider this simple class:
>
> pro DOT_ARROW::try ;compile_opt idl2 print,"GOT: ",self->item([1,2,3,4]) end
>
> function DOT_ARROW::item, p return, size(p,/DIMENSIONS) end
>
> pro dot_arrow__define st={DOT_ARROW,$ item:0.0} end
>
> IDL> d=obj_new('dot_arrow') IDL> d->try GOT: 0.00000   0.00000   0.00000   0.00000 ;; RUNS without
> ERROR, but is WRONG!!!
>
> Now with IDL2 enabled (or in IDL 7):
>
> IDL> d->try GOT: 4 ;; RIGHT!!!
>
> IDL 8 has gone one step further than introducing a new ambiguity between "." and "->" in the name of Python/JS-esque
> cosmetics, it's completed that ambiguity -- essentially enforcing it on us -- by making it work both ways. In
> essence, IDL8 is interpreting:
>
> self->item([1,2,3,4])
>
> as
>
> self.item[[1,2,3,4]]
>
> (no indexing error since you can't go out of bounds with an indexing vector by default). If I say "->", I mean
> invoke a method, no matter what. However, as new programmers adopt "." for method invocation, this ambiguity will

> never go away.
>
> And this is not a simple matter of troublesome syntax errors. What's really scary about this is, there are plenty of
> imaginable cases in which a->b(c) is interpreted by IDL8 in a completely different way from earlier versions of IDL,
> yet it can run through without error, silently corrupting your output. There is no warning against or detection of
> the use of identical names for a method-function and a class structure variable.
>
> Chris Torrence noticed this ambiguity could cause trouble while IDL 8 was being designed; I don't recall what the
> final decision was. But in any case, my understanding was this would be a new ambiguity, affecting new code which
> uses the dot operator for method invocation. Never was it discussed that IDL 8 would render inoperable (or worse,
> operable but incorrect) existing, functioning code by re-defining dereferencing post facto. I can scarcely see how
> this marketing-driven syntax change was worth it.
>
> JD

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by David Fanning on Mon, 16 May 2011 21:19:39 GMT
View Forum Message <> Reply to Message

Paul van Delst writes:

> Your post has been saved to my desktop (in 12years, I've only done that 4 times) for immediate reference.

I posted it nearly verbatim in an article on my
web page today, too, in case anyone is out of
desktop space. :-)

It does give you pause, though, doesn't it.

Cheers,

David

--
David Fanning, Ph.D.
Fanning Software Consulting, Inc.
Coyote's Guide to IDL Programming: http://www.idlcoyote.com/
Sepore ma de ni thui. ("Perhaps thou speakest truth.")

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by wlandsman on Mon, 16 May 2011 21:58:33 GMT
View Forum Message <> Reply to Message

On Monday, May 16, 2011 1:27:30 PM UTC-4, JDS wrote:

```
>  Now with IDL2 enabled (or in IDL 7):
>
>  IDL> d->try
>  GOT:           4  ;; RIGHT!!!
>
```

So if compile_opt idl2 is set in our programs then we are OK?   There are several other ambiguities that are fixed by setting compile_opt idl2 and my understanding is that it is sort of de rigueur for IDL analysis.

--Wayne

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by JDS on Mon, 16 May 2011 22:35:02 GMT
View Forum Message <> Reply to Message

My understanding is compile_opt idl2 alleviates this particular undesirable feature of the newly ambiguated syntax, though I'm not eager to go back and insert it for thousands of methods, or develop methods to discover potential brokenness. I suppose talk of enabling idl2 by default died off; I'd be in favor of it. It would break old code, but in a well-controlled and presumably well-advertised way.

By the way, this behavior seems to have been introduced in 8.0.1.   I believe the short lived IDL 8.0 respected the arrow operator.

I discovered it when attempting to debug a piece of older formerly functioning object code. I set a breakpoint, and tried to step into the relevant function method that was returning nonsense (named, in my case 'Width'). I could never get IDL to step into the method, which I eventually realized was because IDL was parsing the method call:

```
  width=self->Width(pt)
```

as the simple array dereference:

```
  width=self.width[pt]
```

In my case, this threw an error, because width was now a two-element vector, and it was supposed to be a scalar. You can imagine many cases in which no error whatsoever is thrown, and, perhaps less commonly, cases where "plausible-but-wrong" values are returned, which, without some diligence, would simply never be discovered.

In principle, IDL could 1) always respect the arrow operator as a method call (in which case only the foolhardy would use ".") and/or 2) detect these conflicts at compile-time or first object-instantiation time, and at least tell the user something useful before arbitrarily picking one interpretation. But neither of those solutions puts the cat back in the bag.

JD

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by penteado on Tue, 17 May 2011 01:09:21 GMT
View Forum Message <> Reply to Message

On May 16, 7:35 pm, JDS <jdtsmith.nos...@yahoo.com> wrote:
> My understanding is compile_opt idl2 alleviates this particular undesirable feature of the newly ambiguated syntax, though I'm not eager to go back and insert it for thousands of methods, or develop methods to discover potential brokenness. I suppose talk of enabling idl2 by default died off; I'd be in favor of it. It would break old code, but in a well-controlled and presumably well-advertised way.
>
> By the way, this behavior seems to have been introduced in 8.0.1.   I believe the short lived IDL 8.0 respected the arrow operator.

I just dug out an old idl 8.0 and can confirm it already had that
behavior. In a more obvious manifestation of the problem:

```
IDL> a={b:0,c:1.0}
IDL> help,a->c(0)
<Expression>    FLOAT     =       1.00000
```

Since there was never any discussion about using -> for structures,
and this only seems to be undesirable, as JD mentioned, to me it looks
like a bug. The main argument against making idl2 assumed by default
was to keep compatibility, so this seems unintended. A side effect of
some other change to the parser, which went unnoticed.

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by JDS on Tue, 17 May 2011 16:17:46 GMT
View Forum Message <> Reply to Message

In my version of IDL 8.0, your example follows, but my example does not. I.e. self->item([1,2,3,4]) still correctly calls the method in IDL 8.0, but no longer does it in 8.0.1. My take is that IDL 8 introduced the syntax ambiguity ("->" and "." interchangeable), then IDL 8.0.1 reversed the precedence in ambiguous cases, now favoring structure/class variable dereferencing over method calling.

BTW, the documentation mentions this interchangeability in the context of method invocation:

  "Beginning with IDL 8.0, you can use the . and -> forms of the operator interchangeably; they are

   equivalent."

In a sense, this bug cannot be fixed, since it is inherent in the choice to make "." mean two things. Unless idl2 is in force, you simply cannot know what I mean by:

```
  d=a.b(c)
```

even (in the case of "b" being both a method name and a class variable), at runtime! ITT could certainly patch "->" to avoid breaking old code, but new code will always have this potential for silent brokenness (unless people shun "."). What's interesting is the main concern was putting off new users with meaningless syntax error messages. This example shows a much more problematic issue arises.

One possible fix would be to make array subscripting usage following the "." operator implicitly require brackets "[]". This would break old code like c=a.b(4), but leave intact other uses of parentheses for array indexing. I'd call this a "partial idl2 requirement".   It still leaves more than a year's worth of IDL versions in use silently breaking old code.

Just to make it ridiculously obvious to everyone:

```
;+++++++++++++++++++++++
;  failure__define.pro
pro Failure::Explode
  prevent=[1b]  ;; Alway prevent the explosion, no matter what!
  if self->Prevent_Explosion(prevent) then $
     print, 'Explosion averted, go in peace.' $
  else print, '!BOOM! Nuclear war initiated.'
end

function Failure::Prevent_Explosion, confirm
  return, keyword_set(confirm[0])
end

pro failure__define
  st={FAILURE,$
      prevent_explosion: 0b}
end
;+++++++++++++++++++++++
```

```
IDL v5 - v7:
IDL> f=obj_new('failure')
IDL> f->Explode
Explosion averted, go in peace.
```

```
IDL >v8:
IDL> f=obj_new('failure')
IDL> f->Explode
!BOOM! Nuclear war initiated.
```

JD

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by penteado on Tue, 17 May 2011 17:37:45 GMT
View Forum Message <> Reply to Message

On May 17, 1:17 pm, JDS <jdtsmith.nos...@yahoo.com> wrote:
> In my version of IDL 8.0, your example follows, but my example does not. I.e. self->item([1,2,3,4]) still correctly calls the method in IDL 8.0, but no longer does it in 8.0.1. My take is that IDL 8 introduced the syntax ambiguity ("->" and "." interchangeable), then IDL 8.0.1 reversed the precedence in ambiguous cases, now favoring structure/class variable dereferencing over method calling.
>
> BTW, the documentation mentions this interchangeability in the context of method invocation:
>
> "Beginning with IDL 8.0, you can use the . and -> forms of the operator interchangeably; they are
> equivalent."
>
> In a sense, this bug cannot be fixed, since it is inherent in the choice to make "." mean two things. Unless idl2 is in force, you simply cannot know what I mean by:
>
> d=a.b(c)
>
> even (in the case of "b" being both a method name and a class variable), at runtime! ITT could certainly patch "->" to avoid breaking old code, but new code will always have this potential for silent brokenness (unless people shun "."). What's interesting is the main concern was putting off new users with meaningless syntax error messages. This example shows a much more problematic issue arises.
>
> One possible fix would be to make array subscripting usage following the "." operator implicitly require brackets "[]". This would break old code like c=a.b(4), but leave intact other uses of parentheses for array indexing. I'd call this a "partial idl2 requirement". It still leaves more than a year's worth of IDL versions in use silently breaking old code.

This gets confusing easily. If I may expand a bit on JD's example, to
compare the dot with the arrow:

```
;+++++++++++++++++++++++++
; failure__define.pro
pro Failure::Explode_with_arrow
```

```
  prevent=[1b]  ;; Alway prevent the explosion, no matter what!
  if self->Prevent_Explosion(prevent) then $
     print, 'Explosion averted, go in peace.' $
  else print, 'Field used: !BOOM! Nuclear war initiated.'
end

pro Failure::Explode_with_dot
  prevent=[1b]  ;; Alway prevent the explosion, no matter what!
  if self.Prevent_Explosion(prevent) then $
     print, 'Explosion averted, go in peace.' $
  else print, 'Field used: !BOOM! Nuclear war initiated.'
end

function Failure::Prevent_Explosion, confirm
  print,'Got into the method, instead of using the field'
  return, keyword_set(confirm[0])
end

pro failure__define
  st={FAILURE,$
      prevent_explosion: 0b}
end
;+++++++++++++++++++++++++
```

The results:

In IDL 7.1.1, which is how things should still be, to keep
compatibility with old code:

```
IDL> print,!version
{ x86_64 linux unix linux 7.1.1 Aug 21 2009      64      64}
IDL> f=obj_new('failure')
IDL> f->explode_with_arrow
Got into the method, instead of using the field
Explosion averted, go in peace.
IDL> f->explode_with_dot
Field used: !BOOM! Nuclear war initiated.
```

In 8.0 it is broken one way:

```
IDL> print,!version
{ x86_64 linux unix linux 8.0 Jun 18 2010      64      64}
IDL> f=obj_new('failure')
% Compiled module: FAILURE__DEFINE.
IDL> f->explode_with_arrow
Got into the method, instead of using the field
Explosion averted, go in peace.
IDL> f->explode_with_dot
```

Got into the method, instead of using the field
Explosion averted, go in peace.

In 8.1 it is broken the other way:

```
IDL> print,!version
{ x86_64 linux unix linux 8.1 Mar  9 2011      64      64}
IDL> f=obj_new('failure')
IDL> f->explode_with_arrow
Field used: !BOOM! Nuclear war initiated.
IDL> f->explode_with_dot
Field used: !BOOM! Nuclear war initiated.
```

So I would say that what needs to be fixed now is only to make the
arrow mean, always, method invocation. It should never be accepted for
a structure field. The resolution of the ambiguity with the dot got
right in 8.0.1: a field should take precedence over a method.

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by chris_torrence@NOSPAM on Wed, 18 May 2011 22:02:59 GMT

View Forum Message <> Reply to Message

On May 17, 11:37 am, Paulo Penteado <pp.pente...@gmail.com> wrote:
>
> So I would say that what needs to be fixed now is only to make the
> arrow mean, always, method invocation. It should never be accepted for
> a structure field. The resolution of the ambiguity with the dot got
> right in 8.0.1: a field should take precedence over a method.

Hi all,

This is definitely a bug. As Paulo points out, we already fixed the
problem with the "dot". In IDL 8.0.1 and IDL 8.1 the field always
takes precedence over the method, unless "compile_opt idl2" is in
effect. We did this to preserve backwards compatibility - if you had
code that used parentheses, and there was a field with the same name,
then IDL will access the field.

However, it looks like the fix went too far and also included the
arrow. We'll investigate the bug and let you know what we find.

In the meantime, if you run into this problem, you can add
"compile_opt idl2" to resolve the ambiguity.

Thank you.

Cheers,

Chris
ITTVIS

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by JDS on Thu, 19 May 2011 16:20:04 GMT
View Forum Message <> Reply to Message

Thanks, Chris. This also means that continuing to use "->" will offer the safest way forward, since the dual meaning of a.b(c) is implicit and never reported. I think a new user, i.e. those who are most likely to adopt "." for method invocation, is likely to *mean* a->b(c) here, since they quite likely know nothing of the old () form of indexing, and will therefore also rather reasonably presume the difference is between a.b(c) (method call) and a.b[c] (variable subscript).   Without IDL2 in force, and in the likely situation of both a method and class variable named 'b', these are identical.

It would be useful to have a means of enforcing IDL2 across all routines in a file, or even by class (e.g. in class__define), so that you have to include it once and only once.

JD

Subject: Re: New Object Method Invocation Syntax Brokenness
Posted by chris_torrence@NOSPAM on Fri, 10 Jun 2011 20:03:55 GMT
View Forum Message <> Reply to Message

On May 18, 4:02 pm, Chris Torrence <gorth...@gmail.com> wrote:
>
> This is definitely a bug. As Paulo points out, we already fixed the
> problem with the "dot". In IDL 8.0.1 and IDL 8.1 the field always
> takes precedence over the method, unless "compile_opt idl2" is in
> effect. We did this to preserve backwards compatibility - if you had
> code that used parentheses, and there was a field with the same name,
> then IDL will access the field.
>
> However, it looks like the fix went too far and also included the
> arrow. We'll investigate the bug and let you know what we find.
>

Hi all,

We have fixed this bug, and it will be available in the next version
of IDL. In summary:

1. If you use the "arrow", IDL will always call the method (even if
there is a field with the same name).

2. If you use the "dot", and you have both a method and a field with
the same name, IDL will access the field if the compiler cannot tell
whether it is a method call or a field.

Examples:
a. self->something(...)   ; always calls the "something" method,
regardless of what is inside the parentheses
b. self.something[...]   ; always accesses the field
c. self.something(...) {with compile_opt strictarr} ; always calls
the "something" method, regardless of what is inside the parentheses
d. self.something(keyword=5)   ; calls the method because there is a
keyword
e. self.something(0:2)   ; accesses the field because there is an
array range
f. self.something(2,4)   ; Ambiguous! IDL will access the field.

Thanks for reporting this issue.

Cheers,

Chris
ITTVIS