Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by David Fanning on Wed, 11 Sep 2002 15:17:14 GMT
View Forum Message <> Reply to Message

savoie@nsidc.org (savoie@nsidc.org) writes:

> O.k. I'm looking at some pointer weridness. Well, I'm calling it weirdness
> because I obviously don't understand something that is happening. There are
> two examples below.
>
> The first is just two routines. test: creates a pointer, calls changePtr
> with a null pointer as an argument; and changePtr: which just assigns a
> string the the passedPtr. This examples shows that if you pass a pointer to
> a procedure, assign something to that pointer, you can retrieve it after
> exit.
>
>
> The rest of the routines are a simple object with a couple of methods,
> showing exactly the opposite effect. When the object's CHANGEPTR method is
> called, self.myptr doesn't seem to be able to be changed on return.

The problem here has nothing to do with either pointers
or objects. The problem is that structure dereferences
(I.e., self.myptr) are passed by value, whereas passing
the pointer itself (I.e., myptr) is passed by reference.
Procedures can change things that are passed by reference.
They work on *copies* of things that are passed by value.

Since your DOIT method is a self method, you can simple
change it like this:

```
PRO WEIRD::DOIT
  self.myptr =  ptr_new('Why can not I change this?')
END
```

Then, call it like this:

```
   self -> Doit
```

Cheers,

David

--
David W. Fanning, Ph.D.
Fanning Software Consulting, Inc.
Phone: 970-221-0438, E-mail: david@dfanning.com

Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by JD Smith on Wed, 11 Sep 2002 15:55:17 GMT
View Forum Message <> Reply to Message

On Wed, 11 Sep 2002 08:17:14 -0700, David Fanning wrote:

> savoie@nsidc.org (savoie@nsidc.org) writes:
>
>> O.k. I'm looking at some pointer weridness. Well, I'm calling it
>> weirdness because I obviously don't understand something that is
>> happening. There are two examples below.
>>
>> The first is just two routines. test: creates a pointer, calls
>> changePtr with a null pointer as an argument; and changePtr: which just
>> assigns a string the the passedPtr. This examples shows that if you
>> pass a pointer to a procedure, assign something to that pointer, you
>> can retrieve it after exit.
>>
>>
>> The rest of the routines are a simple object with a couple of methods,
>> showing exactly the opposite effect. When the object's CHANGEPTR
>> method is called, self.myptr doesn't seem to be able to be changed on
>> return.
>
> The problem here has nothing to do with either pointers or objects. The
> problem is that structure dereferences (I.e., self.myptr) are passed by
> value, whereas passing the pointer itself (I.e., myptr) is passed by
> reference. Procedures can change things that are passed by reference.
> They work on *copies* of things that are passed by value.
>
>
>
This is the problem, but I think it's also instructive to understand why
exactly it's *not* related to pointers, which otherwise shouldn't care
about by-value or by-reference, since they point to an area of global
heap.

In DOIT, you say:

```
  ptrInside =  ptr_new('Why can not I change this?')
```

With this statement, you are *not*, as you might think, changing the value
of the pointer contained in the argument variable `ptrInside' (which
happens to be the same as the `self.myptr' instance variable). You are

changing the value of the `ptrInside' variable itself. You have assigned
it to a new pointer! Had `self.myptr' already had something in it (i.e.
been a "valid" pointer), you could have said:

```
  *ptrInside='Why can not I change this?'
```

and actually changed the value in `self.myptr'. In your case, the
variable `ptrInside' disappears from the world forever when DOIT returns:
you've just created a memory leak. Note that you can arrange to avoid the
pass-by-value structure problem, with something like:

```
PRO WEIRD::CHANGEPTR
  ptr=self.myptr
  self -> doIt,  ptr
  self.myptr=ptr
END
```

But this doesn't help: either way you risk a memory leak, since only one
of the two pointers created would still be accessible.

David's suggested method is the way to go.

Good luck,

JD

Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by savoie on Wed, 11 Sep 2002 16:02:50 GMT
View Forum Message <> Reply to Message

David Fanning <david@dfanning.com> writes:

> savoie@nsidc.org (savoie@nsidc.org) writes:
>
>> O.k. I'm looking at some pointer weridness. Well, I'm calling it weirdness
>> because I obviously don't understand something that is happening. There are
>> two examples below.
>> <snip/>
>
> The problem here has nothing to do with either pointers
> or objects. The problem is that structure dereferences
> (I.e., self.myptr) are passed by value, whereas passing
> the pointer itself (I.e., myptr) is passed by reference.
> Procedures can change things that are passed by reference.
> They work on *copies* of things that are passed by value.

O.k. I'll agree with that, I actually thought it was being passed by value. But thought, shouldn't a pointer and a copy of a pointer point to the same thing?

Morning coffee hits Aha, but I'm doing is defining what it points to _the first time_ with a copy. This changes the copy, making it a valid pointer, leaving my original pointer alone. Duh.

But if the pointer is already valid, I should be able to dereference the copy and store whatever I want, blissfully ignorant of what it was pointing at thanks to the magic of IDL pointers.

> Since your DOIT method is a self method, you can simple
> change it like this:
>
> PRO WEIRD::DOIT
> self.myptr = ptr_new('Why can not I change this?')
> END
>
> Then, call it like this:
>
> self -> Doit

If Doit didn't have to act on a whole bunch of different internal variables, I could do that.

But it actually does a bunch of repetitive things and is called

```
self->DoIt, self.type1internalPointer, 'type1'
self->DoIt, self.type2internalPointer, 'type2'
self->DoIt, self.typeNinternalPointer, 'typeN'
```

for several different types. I could redesign to an array of internal pointers and an array of types, but since It's already coded the other way....

I can change this Weird::init function to initialize the pointer, and just dereference in the WEIRD::DoIt Function.

```
PRO WEIRD::DOIT,  ptrInside
  *ptrInside =  'Look how I can change this?'
END

;; Don't forget to make your INIT function , a member function
FUNCTION WEIRD::INIT
  self.myPtr = ptr_new('0')
  return,  1
```

END

And hope/trust that IDL is smart enought to not write over the end of the
memory like C would. I vaguely remember a thread about growable arrays that
says I can do this. Anyone think this is a /bad thing/? Rather than just inelegant?

Thanks again for such a fast answer!

Matt Savoie
National Snow and Ice Data Center, Boulder, CO

Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by David Fanning on Wed, 11 Sep 2002 16:37:17 GMT
View Forum Message <> Reply to Message

savoie@nsidc.org (savoie@nsidc.org) writes:

>> Procedures can change things that are passed by reference.
>> They work on *copies* of things that are passed by value.
>
> O.k. I'll agree with that, I actually thought it was being passed by value.
> But thought, shouldn't a pointer and a copy of a pointer point to the same
> thing?

Yes, and I'm sure it does. The problem is not that the pointer
doesn't point to the same thing, the problem is that you can't
get the pointer that is pointing to the same thing back to the
calling program! You don't have a passing mechanism in the program
in the way you wrote it.

> *Morning coffee hits* Aha, but I'm doing is defining what it points to _the
> first time_ with a copy. This changes the copy, making it a valid pointer,
> leaving my original pointer alone. Duh.

You must be drinking the Arabian variety. What I'm drinking
isn't strong enough to figure out what this means. :-)

> But if the pointer is already valid, I should be able to dereference the copy
> and store whatever I want, blissfully ignorant of what it was pointing at
> thanks to the magic of IDL pointers.

I *think* this is right, although I'm not convinced I'm
following the logic every step of the way here.

>
>> Since your DOIT method is a self method, you can simple
>> change it like this:
>>
>> PRO WEIRD::DOIT
>> self.myptr = ptr_new('Why can not I change this?')
>> END
>>
>> Then, call it like this:
>>
>> self -> Doit
>
> If Doit didn't have to act on a whole bunch of different internal variables, I
> could do that.
>
>
> But it actually does a bunch of repetitive things and is called
>
> self->DoIt, self.type1internalPointer, 'type1'
> self->DoIt, self.type2internalPointer, 'type2'
> self->DoIt, self.typeNinternalPointer, 'typeN'
>
> for several different types. I could redesign to an array of internal
> pointers and an array of types, but since It's already coded the other way....

Why can't you just pass in the "type" as a parameter? It already
knows about all the pointers. Those are obviously in the self structure.

> I can change this Weird::init function to initialize the pointer, and just
> dereference in the WEIRD::DoIt Function.
>
>
> PRO WEIRD::DOIT, ptrInside
> *ptrInside = 'Look how I can change this?'
> END
>
> ;; Don't forget to make your INIT function , a member function
> FUNCTION WEIRD::INIT
> self.myPtr = ptr_new('0')
> return, 1
> END

If you just want to initialize the pointer (make it valid, but
not pointing to anything in particular), you can do this:

```
self.myPtr = Ptr_New(/Allocate_Heap)
```

Now it is a valid pointer (it can be de-referenced) that points to
an undefined variable.

> And hope/trust that IDL is smart enought to not write over the end of the
> memory like C would. I vaguely remember a thread about growable arrays that
> says I can do this. Anyone think this is a /bad thing/? Rather than just inelegant?

As long as you know what you are doing, even bad things
are sometimes useful.

> Thanks again for such a fast answer!

My pleasure. Did you know you can get service that is twice
as fast for only a third more per month?

Cheers,

David

--
David W. Fanning, Ph.D.
Fanning Software Consulting, Inc.
Phone: 970-221-0438, E-mail: david@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155

Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by Pavel A. Romashkin on Wed, 11 Sep 2002 17:05:58 GMT
View Forum Message <> Reply to Message

David Fanning wrote:
>
> My pleasure. Did you know you can get service that is twice
> as fast for only a third more per month?

Hey, we'd all like that! But then again, 1/3 more than free in this case
is roughly $200 per hour or $25 per line of code, right? :-)

Cheers,
Pavel

Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by Pavel A. Romashkin on Wed, 11 Sep 2002 17:19:42 GMT
View Forum Message <> Reply to Message

savoie@nsidc.org wrote:
>
> *Morning coffee hits* Aha, but I'm doing is defining what it points to _the
> first time_ with a copy. This changes the copy, making it a valid pointer,
> leaving my original pointer alone. Duh.
>
> But if the pointer is already valid, I should be able to dereference the copy
> and store whatever I want, blissfully ignorant of what it was pointing at
> thanks to the magic of IDL pointers.

Precisely. If passed pointer is not valid, the copy means nothing, too,
and is not returned back to the caller. If it is valid, changing the
copy will change the original. You need no objects to illustrate this.
Make a tiny change to your code:

```
*******
PRO CHANGEPTR, passedPtr
on_error, 1 ; Needed because will err on null ptr
  *passedPtr =  'This is weird?'
END
*******
IDL> a = {a:ptr_new(), b:ptr_new(/alloc)}
IDL> changeptr, a.a
% Unable to dereference NULL pointer: PASSEDPTR.
% Error occurred at: CHANGEPTR          7
%                   $MAIN$
% Execution halted at: $MAIN$
IDL> changeptr, a.b
IDL> print, *a.a
% Unable to dereference NULL pointer: <POINTER  (<NullPointer>)>.
% Execution halted at: $MAIN$
IDL> print, *a.b
This is weird?
```

Works just as you described.
Cheers,
Pavel

Subject: Re: Pointer Behavior Objects Vs Plain routines?
Posted by David Fanning on Wed, 11 Sep 2002 17:19:49 GMT
View Forum Message <> Reply to Message

Pavel A. Romashkin (pavel_romashkin@hotmail.com) writes:

> Hey, we'd all like that! But then again, 1/3 more than free in this case
> is roughly $200 per hour or $25 per line of code, right? :-)

I think of giving free advice as something akin to

breathing. If I didn't do it, I'd be playing tennis
even more than I am now. My wife, unfortunately,
feels otherwise. :-(

Cheers,

David

--
David W. Fanning, Ph.D.
Fanning Software Consulting, Inc.
Phone: 970-221-0438, E-mail: david@dfanning.com
Coyote's Guide to IDL Programming: http://www.dfanning.com/
Toll-Free IDL Book Orders: 1-888-461-0155
