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
