Subject: Re: Object overhead
Posted by Craig Markwardt on Sat, 22 Sep 2001 14:50:46 GMT
View Forum Message <> Reply to Message

"Pavel A. Romashkin" <pavel.romashkin@noaa.gov> writes:

> I noticed that the size of an object array, where each object only has
> one empty poiter field, is 4 times larger than the size of an empty
> pointer array of the same length. PTRARR(1000), when saved, takes about
> 44 Kb. OBJARR(1000) of {junk, data:PTR_NEW()} takes 176 Kb. Why is that?
> Also, it takes twice the time (0.79 vs 0.44 s) to save an object array
> than the pointer array. It can not have to do with the volume of saved
> data because it takes 0.05 s to save an 80 Kb FLTARR(20000).

Hi Pavel--

Since I've mucked around with the format of save files, and in fact made a library to manipulate them, I think I can try to answer your questions.

Saving a FLTARR() is fast because IDL can simply write out a block of floating point numbers in one operation. [after converting to standard endian-ness, of course]

Saving a PTRARR() is slower, because IDL must scan through the entire array looking for non-null pointers. You actually gave an example that was too trivial. If you tried populating your PTRARR(1000) with values, you will find that the save file becomes much larger and take much longer to create. That's because IDL must save each heap variable separately. This is invisible to you, but it happens.

Finally, an array of objects is more complicated still. Objects are really structures. Since structures are more or less infinitely flexible, much more complicated programming is used to save them, and there is extra metadata information related to structures, that must also be saved.

Vector support for structures and pointers has never been as strong as for simple data types, and this shows in how they are save as well.

Hope this helps!
Craig

--
---------- ----------
Craig B. Markwardt, Ph.D. EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
---------- ----------

Subject: Re: Object overhead
Posted by Pavel A. Romashkin on Mon, 24 Sep 2001 15:32:18 GMT
View Forum Message <> Reply to Message

Thanks, Craig!
This kind of helps. Kind of, because you confirm that objects will then always have a limitation: if you have repetitive data that would benefit from objectifying, think twice if you are going to have a lot of them, or you will slow down to a crawl. However, for small numbers, the convenience of objects can't be beat.
Thanks again,
Pavel
P.S. When I say "a lot of them", I think it depends on what you are running on. For my G4, I notice a difference when the number of data objects reaches 5k. And I hoped to be able to convert over 100k... I think I can come up with a way to use objects for processing but store and save data as simple arrays.

Craig Markwardt wrote:
>
> Hi Pavel--
>
> Since I've mucked around with the format of save files, and in fact
> made a library to manipulate them, I think I can try to answer your
> questions.
>
> Saving a FLTARR() is fast because IDL can simply write out a block of
> floating point numbers in one operation. [after converting to
> standard endian-ness, of course]
>
> Saving a PTRARR() is slower, because IDL must scan through the entire
> array looking for non-null pointers. You actually gave an example
> that was too trivial. If you tried populating your PTRARR(1000) with
> values, you will find that the save file becomes much larger and take
> much longer to create. That's because IDL must save each heap
> variable separately. This is invisible to you, but it happens.
>
> Finally, an array of objects is more complicated still. Objects are
> really structures. Since structures are more or less infinitely
> flexible, much more complicated programming is used to save them, and
> there is extra metadata information related to structures, that must
> also be saved.
>
> Vector support for structures and pointers has never been as strong as
> for simple data types, and this shows in how they are save as well.
>
> Hope this helps!
> Craig
>

> --
> ---------- ----------
> Craig B. Markwardt, Ph.D. EMAIL: craigmnet@cow.physics.wisc.edu
> Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
> ---------- ----------

Subject: Re: Object overhead
Posted by John-David T. Smith on Mon, 24 Sep 2001 17:26:05 GMT
View Forum Message <> Reply to Message

"Pavel A. Romashkin" wrote:
>
> I noticed that the size of an object array, where each object only has
> one empty poiter field, is 4 times larger than the size of an empty
> pointer array of the same length. PTRARR(1000), when saved, takes about
> 44 Kb. OBJARR(1000) of {junk, data:PTR_NEW()} takes 176 Kb. Why is that?
> Also, it takes twice the time (0.79 vs 0.44 s) to save an object array
> than the pointer array. It can not have to do with the volume of saved
> data because it takes 0.05 s to save an 80 Kb FLTARR(20000).
> I thought of converting a data processing program to objects but found
> that storage space required almost doubles (well, partially due to some
> redundancy in object fields) and save/restore time increased tenfold.
> For storing many thousands of data objects, this matters.
> Thank you,
> Pavel

Why don't you compare:

ptrarr(1000) to {junk_struct,data:ptrarr(1000)} to obj_new('junk') with

```
pro junk__define
  st={JUNK,data:ptrarr(1000)}
}
```

I'd suspect they'll be close, especially the last two. The basic issue
is that arrays *of* structures, and arrays *in* structures must
necessarily receive quite separate treatment, because structures are
themselves such complex beasts.

Also, as Craig pointed out, this is an unfair test in itself, since the
overhead of pointer arrays will be seen only when you actually attach
them to some data, and will then likely dominate performance and size.
As an example of this for the size issue, I tried:

```
IDL> a=ptrarr(1000) & for i=0,999 do *a[i]=findgen(3*i+1)
```

saving "a" yields 6107296 bytes on disk.

```
IDL> a=replicate({JUNK,data:ptr_new()},1000)
IDL> for i=0,999 do a[i].data=ptr_new(findgen(3*i+1))
```

which is 6107340 bytes on disk. Almost exactly the same! Then I tried:

```
function KNUJ::Init, data
  self.data=ptr_new(data)
  return,1
end

pro KNUJ__define
  struct={KNUJ,data:ptr_new()}
end

IDL> a=objarr(1000) & for i=0,999 do a[i]=obj_new('KNUJ',findgen(3*i+1))
```

which is 6239328 on disk.

I.e., around a 2% savings for the first two methods. Not terribly meaningful differences.

The only conclusion you can draw is that pointer array members carry about 45
bytes of meta-data each, and objects (and probably structures) carry about 180 bytes of meta-data each.

As an example, take structures. All C types of IDL variables are documented in external/export.h. A bit of piecing together reveals an array of structures looks like:

```
typedef struct {              /* IDL_VARIABLE definition */
  UCHAR type;                   /* Type byte */
  UCHAR flags;                  /* Flags byte */
  IDL_ALLTYPES value;
} IDL_VARIABLE;
```

A generic variable, not carrying too much baggage, about 10 bytes.

An array looks like:

```
typedef struct {              /* Its important that this block
                              be an integer number of longwords
                              in length to ensure that array
                              data is longword aligned.  */
  IDL_MEMINT elt_len;         /* Length of element in char units */
  IDL_MEMINT arr_len;          /* Length of entire array (char) */
  IDL_MEMINT n_elts;          /* total # of elements */
```

```
  UCHAR *data;                /* ^ to beginning of array data */
  UCHAR n_dim;                /* # of dimensions used by array */
  UCHAR flags;                /* Array block flags */
  short file_unit;            /* # of assoc file if file var */
  IDL_ARRAY_DIM dim;          /* dimensions */
  IDL_ARRAY_FREE_CB free_cb;  /* Free callback */
  IDL_FILEINT offset;         /* Offset to base of data for file var*/
  IDL_MEMINT data_guard;      /* Guard longword */
} IDL_ARRAY;
```

Each structure itself looks like:

```
typedef struct {              /* Reference to a structure */
  IDL_ARRAY *arr;             /* ^ to array block containing data */
  struct _idl_structure *sdef;  /* ^ to structure definition */
} IDL_SREF;
```

which is an array and the definition (tag names, etc.) The sdef field is "a pointer to an opaque IDL structure definition", i.e. they ain't gonna tell you.

What is the native size of a mostly empty IDL structure? Even without knowing the size of _idl_structure, we already have the following sizes (in bytes)

```
IDL_VARIABLE:     20
IDL_ARRAY:        64
IDL_SREF:         8
IDL_StructDefPtr: 4
++++++++++++++++++++++
              96 bytes + (_idl_structure)
```

Each tag name is associated with a struct of type IDL_STRUCT_TAG_DEF (16 bytes), so we're up to 112 bytes for at least one tag, not to mention the space for the tag name itself. Add in a few more pieces of meta-data inside of _idl_structure, and all of a sudden 180 bytes with 1 tag and without any data at all isn't inconceivable. So the moral of the story: IDL structures *weigh* much more than IDL scalars.

If you have ~100 bytes of data or so per entity, and can handle the loss of flexibility, you'll realize real space savings avoiding all the extra cruft attached to structures. If you have much more data per storage entity (struct, pointer array member, etc.), as in the above example, the space savings will be marginal.

JD
