Subject: Operator Precedence Tutorial
Posted by JD Smith on Fri, 14 Nov 2003 21:11:05 GMT
View Forum Message <> Reply to Message

Precedents, presidents, prescience. Ever wondered what manner of otherworldly intervention is required to work comfortably with deeply nested data structures made from arrays, pointers, and structure variables in IDL? The secret lies in knowing IDL's true operator precedence, which in practical terms is to say the order in which it evaluates combinations of subscripting, pointer dereferencing, structure dereferencing, and parenthetical grouping. The new IDLv6 manual finally has a complete operator precedence table, and it looks like this:

```
First (highest) ( ) (parentheses, to group expressions)
  [ ] (brackets, to concatenate arrays)
Second          .   (structure field dereference)
  [ ] (brackets, to subscript an array)
  ( ) (parentheses, used in a function call)
Third           *   (pointer dereference)
  ^   (exponentiation)
  ++  (increment)
  --  (decrement)
Fourth          *   (multiplication)
  # and ## (matrix multiplication)
  /   (division)
  MOD (modulus)
Fifth           +   (addition)
  -   (subtraction and negation)
  <   (minimum)
  >   (maximum)
  NOT (Bitwise negation)
Sixth           EQ  (equality)
  NE  (not equal)
  LE  (less than or equal)
  LT  (less than)
  GE  (greater than or equal)
  GT  (greater than)
Seventh         AND (Bitwise AND)
  OR  (Bitwise OR)
  XOR (Bitwise exclusive OR)
Eighth          &&  (Logical AND)
  ||  (Logical OR)
  ~   (Logical negation)
Ninth           ?:  (conditional expression)
```

We'll concentrate on using '[]', '*', '.', and '()' together to work with deeply nested data structures. Let's see how it plays out in a few examples.

```
IDL> a={b:findgen(5)}
IDL> print,a.b[2]
      2.00000
```

Note that structure dereferencing and array indexing are at the same level, so whichever comes first takes precedence.

```
IDL> a={b:ptr_new(findgen(5))}
IDL> print,*a.b
      0.00000      1.00000      2.00000      3.00000      4.00000
IDL> print,*a.b[1]
% Subscript range values of the form low:high must be >= 0, < size, with low
  <= high: <No name>.
% Execution halted at: $MAIN$
```

Oh my, that's not what we wanted. Pointer dereference is lower down the list than both structure dereference and subscripting, so we must wield the superlative primacy of the parentheses to make '*' behave:

```
IDL> print,(*a.b)[1]
      1.00000
```

Notice that we didn't say (*a).b[1], since 'a' isn't itself a pointer. But what if we have:

```
IDL> a=ptr_new({b:findgen(5)})
IDL> print,(*a).b[1]
      1.00000
```

Note that, unlike in C, pointer dereferencing is (relatively) low on the precedence stack, which is why:

```
IDL> print,*a.b
% Expression must be a structure in this context: A.
% Execution halted at: $MAIN$
```

doesn't work in this case. Sometimes we have arrays of structures or pointers:

```
IDL> a=replicate({b:findgen(5)},3)
IDL> print,a[0].b[1]
      1.00000
```

Here we index the array of structures first, then dereference the structure, then index the resulting array. Since these are all at the same level of precedence, no parentheses are needed.

Let's make it a bit harder:

```
IDL> a=replicate({b:ptr_new(findgen(5))},3)
IDL> print,*a[1].b[0]
      0.00000      1.00000      2.00000      3.00000      4.00000
```

That's not what we expected, yet no error was issued! Beware: you can get yourself in real trouble with [0]. Remember that '*' has the lowest precedence in that statement. So what's being dereferenced is a[1].b[0], which, in IDL parlance, is the same as a[1].b, which is a scalar pointer, pointing to an array, which was dutifully printed. The reason no error was issued? IDL always lets you "index" a scalar with 0, be it an integer, float, pointer, etc. Had we said:

```
IDL> print,*a[1].b[1]
% Subscript range values of the form low:high must be >= 0, < size, with low
  <= high: <No name>.
% Execution halted at: $MAIN$
```

we would have discovered our mistake sooner. What we want is (you guessed it):

```
IDL> print,(*a[1].b)[1]
      1.00000
```

By now you should be noticing the pattern in how and where I use '()' to group '*' with expressions: I always combine it with the entire sub-expression (nor more, no less) which is itself a pointer; here 'a' is not a pointer, but an array of structures, and a[1] is just a structure -- a[1].b is the pointer we're after, and thus gets grouped with '*'.

We can get arbitrarily complex:

```
IDL> a=replicate({b:ptr_new([ptr_new(findgen(5,2)), $
             ptr_new({c:replicate({d:findgen(5)},2)})])},3)

IDL> print,(*(*a[0].b)[1]).c[1].d[3]
      3.00000
```

Not that you'd ever want to build structures this deep, but isn't it nice to know you can? Notice again the familiar pattern -- grouping '*' with a[0].b, which is a pointer, and with (*a[0].b)[1], which is also a pointer:

```
IDL> print,a[0].b,(*a[0].b)[1]
<PtrHeapVar7><PtrHeapVar6>
```

One way to approach problems like this is to write it all down as if precedence doesn't matter, then go back and group '*' with its target(s). '[]' and '.' will never get in each other's way, since they associate left to right, and have the same precedence, i.e. you'll never have to say (a[0]).b or the like -- '*' is the real trouble-maker. Printing out subexpressions along the way to make sure you know what you're grouping also works well (which, by the way, is infinitely easier with IDLWAVE, with which you can drag-print sub-expressions without interrupting the process of building commands right on the command line). And for commonly used data structures, a few well designed functions or class methods which pull out the pieces you want can obviate all these finger gymnastics.

Also note that since all other operators are effectively below '[]','.','*' in precedence, any arbitrarily complex expression composed of these can be treated just like a simple variable, e.g.:

```
IDL> print,++(*(*a[0].b)[1]).c[1].d[3]^2
      16.0000
```

which is equivalent to:

```
IDL> print,++var^2
```

Also remember that the only occasions where you don't need to explicitly group '*' with its target is when _everything_ after it is its target:

```
IDL> a={b:{c:[ptr_new(5.),ptr_new(6.)]}}
IDL> print,*a.b.c[0]
      5.00000
```

which, using our recipe, could have been grouped with superfluous ()'s as:

```
IDL> print,(*a.b.c[0])
      5.00000
```

But that just wouldn't be elegant ;).
