Variable length arrays in RPG

One of the new additions to RPG in *IBM i* 7.4, and not to the equivalent TR of 7.3, was the ability to have variable length arrays.

Having read the documentation, and having had a chance to "*play*", my findings mean I can divide this post three sections:

1. Setting the number of elements in the array
2. Expanding number of elements
3. Compile time array

Before I get started I want to show what the maximum size of an array, this number has not changed since at least *IBM I* 7.2 . It is not the number of elements in the array that is the limit, it is the total size of the array. The array must not exceed 16,773,104 bytes. For example, this is valid as an array of 16,773,104 elements of one character is valid.

```
dcl-s Array1 char(1) dim(16773104) ;
```

If I increase it by one byte the program will not compile:

```
000500 dcl-s Array2 char(1) dim(16773105) ;
======>                          aaaaaaaa
*RNF3452 20 a   000500  The parameter for keyword DIM is not
                        valid; defaults to 1.
```

Or increase the size of the individual elements to 10:

```
000800 dcl-s Array3 char(10) dim(16773104) ;
======>      aaaaaa
*RNF0376 30 a   000800  Total length 167731040 of data item
                        ARRAY3 exceeds 16,773,104 bytes.
```

If I reduce the number of elements by a factor of ten, then the array is valid.

```
dcl-s Array4 char(10) dim(1677310) ;
```

This is just *for your information*, I am **not** encouraging anyone to create arrays of that size.

Setting the number of elements in the array

When I define my array the format of the `DIM` keyword is different:

```
dcl-s ArrayVar char(1) dim(*var:1000) ;
```

Rather than give a just number for the number of elements I have the `*VAR` value followed by the maximum number of elements the array can have. I chose 1,000 just

because it looked like a nice round number to me, you can pick any number for your own array.

As I have not set the array to any number of elements if I display the number of elements in the array there are zero:

```
dsply ('1. Var = ' + %char(%elem(ArrayVar)) ) ;

> DSPLY  1. Var = 0
```

If I try to move a value into one of the elements of the array I get an error, which is not a surprise as the array has zero elements.

```
ArrayVar(1) = '1' ;

RNQ0121: An array index is out of range (C G D F).
```

To set the size of the array I use the `%ELEM` built in function, then I fill the array up to that number of elements:

```
%elem(ArrayVar) = 5 ;

for Counter = 1 to %elem(ArrayVar) ;
  ArrayVar(Counter) = %char(Counter) ;
endfor ;

dsply ('3. Var = ' + %char(%elem(ArrayVar)) ) ;
dsply ('4. Var(5) = <' + ArrayVar(5) + '>') ;

> DSPLY  3. Var = 5
> DSPLY  4. Var(5) = <5>
```

Then I can just reset the array back to having zero elements:

```
%elem(ArrayVar) = 0 ;
dsply ('5. Var = ' + %char(%elem(ArrayVar)) ) ;

> DSPLY  5. Var = 0
```

What happens to the values in that I placed in the array when I increase the number of elements back to five? The values in the array's elements were cleared.

```
%elem(ArrayVar) = 5 ;
dsply ('6. Var = ' + %char(%elem(ArrayVar)) ) ;
dsply ('7. Var(5) = <' + ArrayVar(5) + '>') ;

> DSPLY  6. Var = 5
> DSPLY  7. Var(5) = < >
```

I know it goes without saying, but if I try to increase the number of elements to greater than the 1,000 maximum I defined in the array's definition I get a compile error:

```
%elem(ArrayVar) = 1001 ;

RNF7563: The expression is not valid for assignment to
         built-in function %LEN or %ELEM.
```

Sometimes IBM adds something I consider bizarre, the following example is one of those.

```
dcl-s ArrayVar2 char(1) inz('*') dim(*var:1000) ;
```

In this array I want to initialize every element with an asterisk (`*`). If I add ten elements to this array then all ten contain an asterisk:

```
%elem(ArrayVar2) = 10 ;
dsply ('1. Var2(10) = <' + ArrayVar2(10) + '>') ;

> DSPLY  1. Var2(10) = <*>
```

What if I now want to have another ten elements, but not have them initialized with the asterisk.

```
%elem(ArrayVar2) = 0 ;
%elem(ArrayVar2:*keep) = 20 ;
dsply ('2. Var2(20) = <' + ArrayVar2(20) + '>') ;

> DSPLY  2. Var2(20) = < >
```

In the second line of code the `%ELEM` BiF has a second parameter, `*KEEP`, this stops the new elements, 11-20, from being initialized with the asterisk. Why you want to initialize the elements in an array with one value and then not want to do it for others is, well, beyond me.

Expanding number of elements

To use an error with an "expanding" number of elements I need to use the `*AUTO` value in the `DIM` of the array's definition:

```
dcl-s ArrayAuto char(1) dim(*auto:1000) ;
```

The array starts off with zero elements…

```
dsply ('1. Auto = ' + %char(%elem(ArrayAuto)) ) ;

> DSPLY  1. Auto = 0
```

And I can add elements to the array just by referencing them, no need to allocate a number of elements as I did before. For example:

```
for Counter = 1 to 5 ;
  ArrayAuto(Counter) = %char(Counter) ;
endfor ;
dsply ('2. Auto = ' + %char(%elem(ArrayAuto)) ) ;

> DSPLY  2. Auto = 5
```

As before I can set the number of array elements to another number just by using the `%ELEM` BiF:

```
%elem(ArrayAuto) = 0 ;
dsply ('3. Auto = ' + %char(%elem(ArrayAuto)) ) ;

> DSPLY  3. Auto = 0
```

A new keyword that can be used with the expanding array has been introduced, *NEXT. Rather than having to give the number of the next available array element I can use, the *NEXT does that for me. For example:

```
%elem(ArrayAuto) = 0 ;

for Counter = 1 to 5 ;
  ArrayAuto(*next) = %char(Counter) ;
endfor ;
dsply ('8. Auto = ' + %char(%elem(ArrayAuto)) ) ;
dsply ('9. Auto(5) = <' + ArrayAuto(5) + '>') ;

> DSPLY  8. Auto = 5
> DSPLY  9. Auto(5) = <5>
```

The statement in the For group uses the *NEXT for the next array element to use. At the end of the For group there are five elements in the array.

The most common types of arrays I use are data structure arrays. Can the number the elements in these expand when used?

```
01  dcl-ds Data qualified dim(*auto:9999) ;
02    Name varchar(40) ;
03    PlaceOfBirth varchar(50) ;
04  end-ds ;
05  dcl-s Rows uns(5) ;

06  exec sql DECLARE C0 CURSOR FOR
07             SELECT RTRIM(LNAME) || ', ' || FNAME,
08                    IFNULL(PLACEBIRTH,'*UNKNOWN')
09               FROM PERSON
10              ORDER BY LNAME,FNAME
11                FOR READ ONLY ;

12  exec sql OPEN C0 ;

13  dsply ('1. Elements = ' + %char(%elem(Data)) ) ;

14  exec sql FETCH C0 FOR 9999 ROWS INTO :Data ;

15  exec sql GET DIAGNOSTICS :Rows = ROW_COUNT ;

16  exec sql CLOSE C0 ;

17  dsply ('2. Rows fetched = ' + %char(Rows) ) ;
18  dsply ('3. Elements = ' + %char(%elem(Data)) ) ;
```

Line 1: The DIM keyword contains the *AUTO and the maximum number of elements I want this array to have.

Line 5: This variable will be used to contain the number of rows retrieved by the SQL statement. I have defined this variable as an unsigned integer, as I cannot retrieve a fraction of a row or a negative number of them.

Lines 6 – 11: The definition of the cursor I will be using to fetch rows from the table. I am using the `IFNULL`, line 8, in the Place of Birth column to convert any columns that have a value of null to something more user friendly.

Line 12: Open the cursor.

Line 13: I want to know the number of array elements in my data structure just before I fetch the results.

Line 14: I perform the fetch for the maximum number of possible array elements. In the past I have used a variable here that was defined with the number of data structure array elements, but in this case it will not work. You will see why in a moment. Therefore, I have had to "hard code" the number of elements, 9,999.

Line 15: I use `GET DIAGNOSTICS` to retrieve the number of rows fetched.

Line 16: Close the cursor.

Lines 17 and 18: Display the number of rows fetched and the number of elements in the array.

When I run this program I see:

```
DSPLY  1. Elements = 0
DSPLY  2. Rows fetched = 41
DSPLY  3. Elements = 41
```

This shows that before the fetch statement the array has no elements. 41 rows were fetched into 41 elements of the array. The allocating of the elements for the results was done automatically, without me having to do anything.

Compile time array

Compile time arrays, you know those ones where you give the list of values at the bottom of the program.

```
dcl-s ArrayCt1 char(1) ctdata dim(5) ;

** ArrayCt1
1
2
3
4
5
```

It has always been one of my *pet peeves* that the RPG compiler knows how many items I have in my compile time array, but I always have to give it in the definition statement.

Now that *peeve* can be laid to rest, as I can now define an compile time array thus:

```
dcl-s ArrayCt1 char(1) dim(*ctdata) ;
dsply ('1. Ct1 = ' + %char(%elem(ArrayCt1)) ) ;
```

```
> DSPLY  1. Ct1 = 5
```

Now I can add to or delete elements from the compile time array and not have to count how many elements there are and change the definition.

If you do define compile time tables this new way you will need to place them in your code before any other compile time arrays defined in the previous way. If not your program will not compile.

```
RNF0203: DATA FOR ARRAYS AND TABLES WITH DIM(*CTDATA) MUST
         PRECEDE ALL OTHER COMPILE-TIME DATA.
```

- Varying dimension arrays
- DIM(*CTDATA)