System i Developer

Advanced Data Structures

Jon Paris

Jon.Paris @ Partner400.com
www.Partner400.com
www.SystemiDeveloper.com

Notes

About Me:

I am the co-founder of Partner400, a firm specializing in customized education and mentoring services for IBM i (AS/400, System i, iSeries, etc.) developers. My career in IT spans 45+ years including a 12 year period with IBM's Toronto Laboratory.

Together with my partner Susan Gantner, I devote my time to educating developers on techniques and technologies to extend and modernize their applications and development environments. Together Susan and I author regular technical articles for the IBM publication, IBM Systems Magazine, IBM i edition, and the companion electronic newsletter, IBM i EXTRA. You may view articles in current and past issues and/or subscribe to the free newsletter at: www.IBMSystemsMag.com. We also write frequently for IT Jungle's RPG Guru column (www.itjungle.com).

We also write a (mostly) monthly blog on Things "i" - and indeed anything else that takes our fancy. You can find the blog here: ibmsystemsmag.blogs.com/idevelop/

Feel free to contact me any time: Jon.Paris @ partner400.com

Agenda

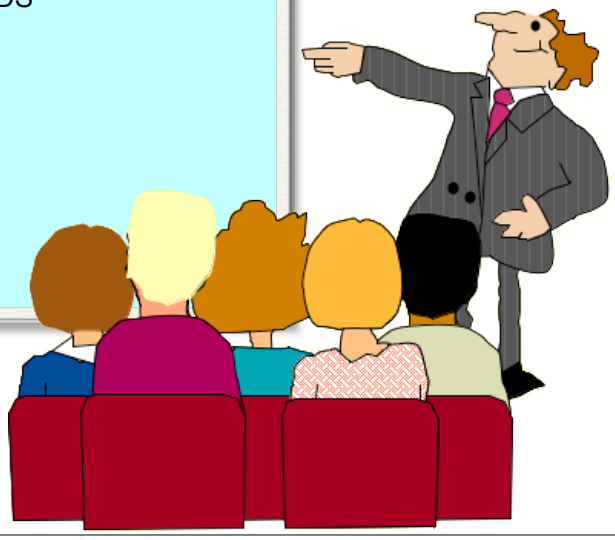
Partner400

What do I mean by "Advanced" ?

DS stuff that you might not know
- No-length fields
- No-name fields
- Incorporating external fields into a DS
- Group fields
- Mapping Indicators to Names

Data Structures
- V5R2 - LikeDS and LikeRec
- Templates

Subfile Sort Example
- Using many of these capabilities



Notes

Partner400

Many of the ideas described in this presentation came about either because of our personal dislike of certain things (like Compile time arrays) and seeking an alternative. Sometimes we discovered them as a direct result of responding to customers asking us how to achieve a specific task.

Unnamed Fields and Overlaying a DS

DS subfields do not need to be named

- But they can still have INZ values!

Makes a great alternative to using compile-time data

- Initialize the data near the array definition itself
- No need to chase to the end of the source member
- The fields can have names if you wish but they do not have to

Our passionate dislike of compile-time data led us to this technique

- It makes life much simpler when you want to see the values used to intialize an array.

```
D Messages        DS
D                                20a   Inz('Invalid Item Code')
D                                20a   Inz('Too many selections')
D                                20a   Inz('Item Code required')
D                                20a   Inz('Huh?')

D    Msg                         20a   Overlay(Messages) Dim(4)
```

Notes

This is a technique we came up with several years ago in response to our frustrations in dealing with programs that used compile time data.

Frustration? Imagine working through a program and encountering the use of an array element. You then move to the data definitions to see what the array looks like (and hopefully find a comment explaining it). What you discover is that it is in fact a compile time array - and you now have to get to the opposite end of the program to find out what the array contains!

This technique avoids that and places the values alongside the data definitions.

Unnamed Fields and Overlaying a DS

Partner400

Differences between fixed and free-form versions:

- You must use the placeholder *N instead of leaving the name blank
  - But it can still have an INZ value
- In Free-form the Overlay keyword can only apply to fields
  - It cannot be applied to the DS name as in fixed-form
- You must use the POS(n) keyword to position the redefinition

```
dcl-ds Messages;
  *n          Char(20) Inz('Invalid Item Code');
  *n          Char(20) Inz('Too many selections');
  *n          Char(20) Inz('Item Code required');
  *n          Char(20) Inz('Huh?');

  Msg         Char(20) Dim(4) Pos(1);
end-ds;
```

Free Form Version

Notes

Partner400

As you can see in the free-form version, there are two major differences.

First the name cannot be completely omitted - instead you have to use the *N placeholder. Without this, the compiler would attempt to treat the data type definition of the field as its name because the name is the first entry on the line. This was not a problem with the fixed-form version as the name had to be placed in a "box" and could never be confused with anything else.

The second is that in free-form the Overlay keyword cannot be used against a DS name - only against field names. Apparently IBM found that allowing the use of the DS name caused a lot of misunderstandings and errors. So they decided to revert to the original intent for the keyword (to provide a means to redefine a field into its component parts) with free-form and to add the POS(n) keyword. POS(n) allows you to specify that the field starts in a specific position within the DS. In many cases this will be POS(1) - but as you will see later there are other useful things that you can use POS for and it is far less typing, and less error prone, than using the OVERLAY alternative.

No-Length Subfields

Want to access fields in a record as an array?

- The "secret" is to place them in a DS
- This works even if the fields are not contiguous in the record

No length definition is required

- The compiler uses the length supplied by the database

Use the POS keyword to cause the array to overlay the DS

- Notice the use of the LIKE keyword

```
dcl-f MthSales;

dcl-ds
  SalesData;
    Q1;
    Q2;
    Q3;
    Q4;

    SalesForQtr  Like(Q1)  Pos(1)  Dim(4);
end-ds;
```

Free Form Version

DDS

```
R MTHSALESR
  CUSTOMER          4
  STREET           32
  CITY             24
  STATE             2
  DIVISION          2
  Q1                7  2
  Q2                7  2
  Q3                7  2
  Q4                7  2
K CUSTNO
```

Notes

The inspiration for this example comes from a commonly asked question on RPG programming lists: “How do I directly load fields from a database record into an array?” The question normally arises when handling old databases that are not normalized. Typically these are from old S/36 or S/38 applications. The type of record I mean contains a series of related values - for example, sales figures for January, February, ..., December.

To make our example fit on the page we're not going to show 12 months, because it wouldn't fit on the chart! Hopefully the example of sales figures for four quarters will give you the idea of how it all works. The DDS for the physical file is shown. One solution is depicted here. We'll look at a slightly different solution on the next chart. Our objective is to access the individual fields Q1-Q4 as an array of four elements after reading a record from the file.

Notice that we’ve incorporated the Quarterly sales fields into the DS by specifying their names. No length or type definition is required.

No-Length Subfields

Fixed-Form version

- No length definition
- Just name the fields in the correct sequence
- Use the OVERLAY keyword against the DS name

DDS

```
R MTHSALESR
  CUSTOMER          4
  STREET           32
  CITY             24
  STATE             2
  DIVISION          2
  Q1                7  2
  Q2                7  2
  Q3                7  2
  Q4                7  2
K CUSTNO
```

```
D SalesData       DS
D  Q1
D  Q2
D  Q3
D  Q4
D  SalesForQtr                  Overlay(SalesData)
D                               Like(Q1) Dim(4)
```

Notes

This is the right way to handle this requirement and imposes zero performance overhead. But from time to time you may come upon an example that uses the technique shown on the next chart. When you do - FIX IT.

Alternatively you could run away and find a different job where the suicidal tendencies among the programmers are not as prevalent.

Do NOT Use This "Technique"

The idea is to take the address of Q1 and use it to base the array

- It only works if RPG places the Qn fields in consecutive memory locations
  - Sometimes it will work out that way
- But often RPG will not do that
  - And a simple recompile of the program could change this

This is a DANGEROUS technique

- It may work for a while
- But it will break unpredictably at some future time.

```
D SalesForQtr     S                   Based(pQ1)
D                                     Like(Q1) Dim(4)

D pQ1             S                 * Inz(%Addr(Q1))
```

Notes

From time to time you may come upon an example that uses a pointer that contains the address of (in this case) field Q1 and then uses that as the basing pointer for an array - as shown on the chart.

If you see such an example FIX IT !!!

There is ZERO guarantee that fields Q2, Q3 and Q4 follow Q1 in memory - which is what this technique relies upon. A simple recompile of the program could change this if the compiler writers altered the way that data definitions are generated. A simple change to the source code is even more likely to break things - even if the change is unrelated to the array in question.

I'll repeat - THIS IS DANGEROUS - It may work for a while but the only guarantee is that it will break unpredictably at some time. The ONLY way to guarantee that fields occupy contiguous storage is by manually placing them in a DS. Period. End of story.

Group Fields

Sometimes it is convenient to reference a group of fields

- E.g. To group Street, City and State under the name Address.
  - That way they can be manipulated as a single entity

This version of the previous example uses this approach

- Compiler derives the length of QuarterData from the combined lengths of the subfields that OVERLAY it

```
D SalesData     DS
D  Customer
D  Address
D    Street                     Overlay(Address)
D    City                       Overlay(Address: *Next)
D    State                      Overlay(Address: *Next)
D  Division
D  QuarterData
D    Q1                         Overlay(QuarterData)
D    Q2                         Overlay(QuarterData: *Next)
D    Q3                         Overlay(QuarterData: *Next)
D    Q4                         Overlay(QuarterData: *Next)

D  SalesForQtr                  Overlay(QuarterData) Like(Q1) Dim(4)
```

Notes

Note that neither Address or QuarterData have a length, type definition or LIKE keyword. Nor do they exist in any of the program's files. Normally you'd expect such definitions to result in Field Not Defined errors, but it doesn't because the subsequent OVERLAY references inform the compiler that these fields represent a group field.

If you look in detail at QuaterData, you will see that it comprises the fields Q1 though Q4. If you examine the extract from the compiler cross-reference listing below, you'll see that QuarterData is defined as 28 characters long (i.e., the combined lengths of Q1, Q2, Q3 and Q4):

```
Q1                S(7,2)                  14D . . . .
Q2                S(7,2)                  15D . . . .
Q3                S(7,2)                  16D . . . .
Q4                S(7,2)                  17D . . . .
QUARTERDATA       A(28)                   13D . . . .
SALESDATA         DS(34)                  10D . . . .
SALESFORQTR(4)    S(7,2)                  19D . . . .
```

Group Fields - Free Form version

Free Form Version

```
dcl-f MthSales;

dcl-ds SalesData;
  Customer;
  Address;
    Street              Overlay(Address);
    City                Overlay(Address: *Next);
    State               Overlay(Address: *Next);
    Division;
  QuarterData;
    Q1                  Overlay(QuarterData);
    Q2                  Overlay(QuarterData: *Next);
    Q3                  Overlay(QuarterData: *Next);
    Q4                  Overlay(QuarterData: *Next);

  SalesForQtr           Overlay(QuarterData) Like(Q1) Dim(4);
end-ds;
```

Notes

Using SORTA with Group Fields

Want to sort an array on multiple keys?

- Group fields provide an answer
- You can also use qsort - more on this later

ProductData is a group field

- It comprises the fields Name and UnitPrice
- Notice that the DIM is specified at the group level
  - This allows the array to be sorted on any of the subfields
  - The associated data will "follow along" and stay in sync

```
Dcl-DS  ProductInfo;
  ProductData  Dim(1000);
    Name  Char(8)  Overlay(ProductData);
    UnitPrice   Packed(7:2) Overlay(ProductData: *Next);
End-DS;

SortA Name; // Sort Name sequence (Ascend because ?)

SortA(D) UnitPrice; // Sort in descending Unit Price
```

Notes

Remember that when using this technique all of the other fields in the array (i.e. those that are part of the group) will be "pulled along" with their associated values.

Fixed Form version of code:

```
D ProductInfo     DS
  // Note that Dim is specified at the group field level
D   ProductData                        Dim(1000)
D     Name                      20     Overlay(ProductData)
D     UnitPrice                  7p 2 Overlay(ProductData: *Next)


  SortA Name;         // Sort into Name sequence

  SortA(D) UnitPrice;    // Sort in descending Unit Price
```

Prior to V7 if you needed to sort an array with SORTA you could only specify ASCEND or DESCEND on the actual array definition to control the sequence. In V7.1 IBM added the ability to specify the required sort sequence as an operation extender to SORTA. So SORTA(A) will sort the array in ascending sequence and SORTA(D) in descending order. More on this in a moment.

Using SORTA with Sequence Control

Need to sort an array in Ascending OR Descending sequence ?

- The technique below uses twin definitions
  - It has the advantage that it will use high speed%Lookup operations
- You could also use SORTA(A) or SORTA(D)
  - But %Lookup will be much slower

```
Dcl-DS  ProductInfoAsc  Qualified;
  ProductData  Dim(1000) Ascend;
    Name         Char(8)       Overlay(ProductData);
    UnitPrice    Packed(7:2)   Overlay(ProductData: *Next);
End-DS;
// This "Based" version allows the alternate sorting seq.
Dcl-DS ProductInfoDsc  Based(pProductInfoAsc) Qualified;
  ProductData  Dim(1000) Descend;
    Name         Char(8)       Overlay(ProductData);
    UnitPrice    Packed(7:2)   Overlay(ProductData: *Next);
End-DS;

Dcl-S  pProductInfoAsc  Pointer  Inz(%Addr(ProductInfoAsc));

SortA  ProductInfoAsc.Name; // Sort in ascending Name sequence

SortA  ProductInfoDsc.Name; // Sort in descending Name sequence
```

Notes

Note that when using this technique all of the other fields in the array (i.e. those that are part of the group) will be "pulled along" with their associated values.

ASCEND or DESCEND can be specified as normal along with the DIM keyword. So, while you can sort on any of the fields in the group, until V7.1, you could only sort a specific array in ascending OR descending sequence.

In order to allow alternate sequencing you could use a pointer to base a second version of the array as shown in the example below:

```
D ProductInfo     DS
D   ProductData                       Dim(1000) Ascend
D     Name                        8   Overlay(ProductData)
D     UnitPrice                   7p 2 Overlay(ProductData: *Next)

  // Use a Based version of the DS to allow the alternate sorting seq.

D AlternateView   DS                  Based(pProductInfo)
D   ProductDataD                      Dim(1000) Descend
D     NameD                       8   Overlay(ProductDataD)
D     UnitPriceD                  7p 2 Overlay(ProductDataD: *Next)

D pProductInfo    S                *   Inz(%Addr(ProductInfo))
```

Mapping *INnn Indicators to Names

Partner400

Use a pointer to map the standard *IN indicator array

- An easy way to give names to all of your indicators
- Use _nn at the end of the name to document the actual indicator number
  - Thanks to Aaron Bartell for introducing us to this variation

You can also use a group field to map a set of indicators

- Enabling you to clear or set them as a group

```
dcl-s pIndicators     Pointer Inz(%Addr(*In));

dcl-ds DspInd         Based(pIndicators);
// Response indicators
  Exit_03             Ind Pos(3);
  Return_12           Ind Pos(12);
// Conditioning indicators
  ErrorFlags_31_33  Pos(31);
    Error_31            Ind Overlay(ErrorFlags);
    StDateError_32      Ind Overlay(ErrorFlags: *Next);
    EndDateError_33     Ind Overlay(ErrorFlags: *Next);
end-ds;
```

The use of POS makes it more readable than OVERLAY

Notes

Partner400

Note that I used a group field to define the set of indicators 31 - 33 as the single field ErrorFlags. I could also have defined it as being Char(3) Pos(30). This allows me to simply code things like Clear ErrorFlags; I use this technique for subfile control indicators and use constants with names such as DISPLAYSUBFILE and CLEARSUBFILE which are defined as patterns of character 1s and 0s as appropriate. Helps to make the code far more readable.

The "IBM approved" method of naming indicators is the Indicator Data Structure (INDDS) option for files. But that only works with externally described files - so program described printer files cannot take advantage of the capability. Also INDDS creates a separate set of 99 indicators for each structure used. That can be useful but it also means that INDDS can be hard to use in existing code where *INnn indicators are already in use since indicator 99 in a display file that uses INDDS is NOT the same thing and *IN99.

Unlike the INDDS approach, these named indicators DO directly affect the content of their corresponding *IN indicator. So, using the above example, if we code Error = *On then indicator *IN30 was just turned on. This often makes this a better approach for those who use program described (i.e. O-spec) based files rather than externally described printer files.

Those of you who use the *INKx series of indicators to identify function key usage need not feel left out. A similar technique can be used. In this case the pointer is set to the address of *INKA. The other 23 function key indicators are in 23 bytes that follow. IBM have confirmed many times that for RPG IV this will always be the case.

```
dcl-ds FunctionKeys   Based(pFunctionKeys);
   F3_Pressed         Ind Pos(3);
   F12_Pressed        Ind Pos(12);
end-ds;

dcl-s pFunctionKeys   Pointer Inz(%Addr(*InKA));
```

D-Spec Version of Mapping*INnn Indicators

As you can se it is basically the same

- Note I did not include the group field ErrorFlags in this version

As before the *INnn indicators are associated with names

```
D DspInd          DS                    Based(pIndicators)
 // Response indicators
D    Exit_03                       N    Overlay(DspInd: 3)
D    Return_12                     N    Overlay(DspInd: 12)

 // Conditioning indicators
D    Error_31                      N    Overlay(DspInd: 31)
D    StDtErr_32                    N    Overlay(DspInd: 32)
D    EndDtErr_33                   N    Overlay(DspInd: 33)

D pIndicators     S                *    Inz(%Addr(*In))
```

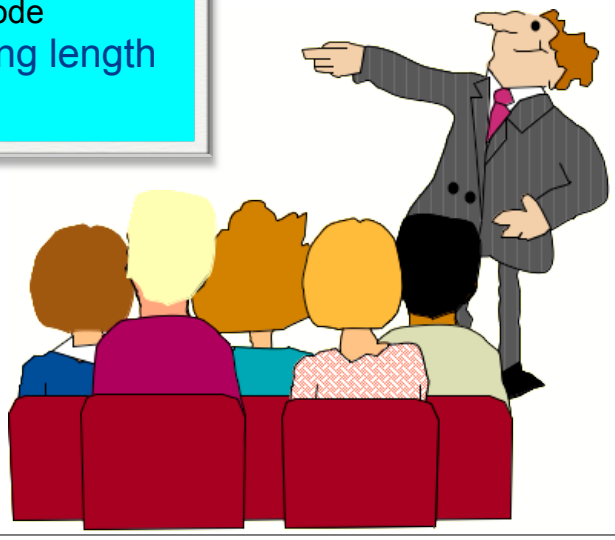
Notes

Use the same basic technique for the *INKx series of indicators to identify function key usage.

Just set the pointer to the address of *INKA. The other 23 function key indicators are in 23 bytes that follow. IBM have confirmed many times that for RPG IV this will always be the case.

Data Defintion Update

V5R2 and Beyond
Data Definition Features
These are essential to processing XML
with RPG's built-in XML-INTO support
And the new DATA-INTO opcode
Plus a quick look at Varying length fields



Notes

V5R2 saw the biggest change in data definition capabilities in the history of RPG. Many new language features are based on these capabilities.

For example, both XML processing and RPG Open Access rely on them. You NEED to understand them. In addition the new DATA-INTO operation is a combination of XMl-INTO and Open Access and so also requires an understanding of these new DS capabilities.

Qualified Data Names & LIKEDS

DS Keyword LIKEDS

- The new DS inherits all the fields of the named DS
- This includes all field names and their size and data type attributes
  - But NOT any DIM or OCCURS specifications on the original DS definition
- Use of LIKEDS implicitly adds the QUALIFIED keyword to the new DS
- Use INZ option *LIKEDS to clone initialization values

Avoids the need to use suffixes or prefixes

- You can now have multiple versions of the fields with the same name

```
Dcl-Ds date;
   year    Zoned(4)   Inz(2003);
   month   Zoned(2)   Inz(01);
   day     Zoned(5)   Inz(01);
 End-Ds;

 Dcl-Ds  orderDate  LikeDS(date)   Inz(*LikeDS);

 If year > baseYear;
    //...
 If orderDate.year > baseYear;
    //...
```

Notes

Previously in RPG one field name = one storage location. As a result if a field name appeared in two different files then reading either file would change the field's content. This often necessitated using prefixes or suffixes on the base name to differentiate between the versions. You might have ARACCNO and CMACCNO for example, both of which are actually the account number (ACCNO). Now that we can use qualified data names in RPG the need for this diminishes significantly. It will take time to adjust the way we deal with files, but in our programs we can now have multiple versions of a field name differentiated by the DS name.

The new keyword LIKEDS causes all fields and their definitions in the original DS to be cloned in the new DS. By definition the new DS is automatically qualified.

In addition there is also a new parameter value allowed on the INZ keyword for data structures defined with the LIKEDS keyword: INZ(*LIKEDS).

This means that any initial values specified in the original data structure should be replicated in the new (LIKEDS) data structure as well. For example, if in our code sample on this chart had an initial value for the Year subfield in Date, such as INZ(2001), that initial value would ONLY be carried over to the OrderDate.Year field IF we added the keyword INZ(*LIKEDS) to the OrderDate D spec.

There is a particularly good discussion and example of this support as it relates to passing a Data Structure as a prototyped parameter in the article by Hans Boldt and Barbara Morris in IBM's iSeries Magazine, May 2001 issue.

```
D date            DS
D   year                        4s 0 Inz(2003)
D   month                       2s 0 Inz(01)
D   day                         5s 0 Inz(01)

D orderDate       DS                 LikeDS(date) Inz(*LikeDS)
```

Qualified Data Names & LIKEDS - more

Free-form works in basically the same way as fixed

- Note that the Dim(10) on the original DS is not cloned
- In this example the parent DS was qualified
  - It can help avoid confusion if all instances of a field name are qualified
- Note the positioning of the array indices
  - They belong with the name of the item that had the Dim statement
  - More on DS arrays later

The DIM here will NOT be cloned - only the subfield definitions

```
dcl-ds date  Dim(10) Qualified;
  year       zoned(4) Inz(2003);
  month      zoned(2) Inz(01);
  day        zoned(5) Inz(01);
end-ds;

dcl-ds orderDate Dim(99) LikeDS(date) Inz(*LikeDS);

if date.year > baseYear;
  ...
if orderDate(1).month = orderDate(2).month;
  ...
```

Notes

The free form implementation is basically the same - just prettier!

In this chart we have also introduced the concept of Data Structure arrays. These are a major improvement on the old Multiple Occurrence Data Structures (MODS) which are still supported for compatibility reasons. We will be going into more detail on DS arrays in a few minutes.

Notice that even though the original array had a DIM statement, this is NOT cloned. The new DS must specify its own DIM if it is to be an array.

Nested DS

The LIKEDS keyword is used to specify the DS to be nested

- Any DS that is to contain a nested DS(s) **must** specify QUALIFIED

To reference the fields requires double qualification

- e.g. InvoiceInfo.MailAddr.City or InvoiceInfo.ShipAddr.State

But wait - there's more !!

- Data Structure arrays !!!!

```
Dcl-Ds  Address;
  Street1  Char(30);
  Street2  Char(30);
  City     Char(20);
  State    Char(2);
  Zip      Zoned(5);
  ZipPlus  Zoned(4);
End-DS;

Dcl-DS  InvoiceInfo  Qualified;
  MailAddr  LikeDS(Address);
  ShipAddr  LikeDS(Address);
End-DS;
```

Notes

This is the fixed form version of these definitions:

```
D Address         DS
D   Street1                    30a
D   Street2                    30a
D   City                       20a
D   State                       2a
D   Zip                         5s 0
D   ZipPlus                     4s 0

D InvoiceInfo     DS                  QUALIFIED
D   MailAddr          LikeDS(Address)
D   ShipAddr          LikeDS(Address)
```

Note that when the LIKEDS keyword is used to reference a DS array, the DIM characteristic is not copied by the LIKEDS. Only the individual components of the DS (fields, conventional arrays and nested DSs) are copied. Look at this example:

```
Dcl-DS  DSArray  Dim(20) Qualified;
   Data  Char(12);

Dcl-DS  NewDS Qualified;
   NewDSArray  LikeDS(DSArray);
```

Even though DSArray is itself an array, when we reference it via the LIKEDS keyword in NewDS, the DIM characteristic is not inherited. In order to establish NewDSArray as a array DS, we explicitly code the DIM keyword, as shown below.

```
Dcl-DS  NewDS Qualified;
   NewDSArray  LikeDS(DSArray)  Dim(20);
```

Data Structure Arrays

The DIM keyword can now also be specified at the DS level

- More useful than MODS since all levels are accessible at once
- Keyword QUALIFIED is also required
  - But we're not quite sure why

Subscripting works in the same way as for individual fields

- Address(5) is the whole of the fifth element of the Address DS array
- Address(5).City is the City field within that fifth element

```
Dcl-Ds  Address  Dim(20) Qualified;
  Street1  Char(30);
  Street2  Char(30);
  City     Char(20);
  State    Char(2);
  Zip      Zoned(5);
  ZipPlus  Zoned(4);
End-DS;

If Address(5).City = 'Rochester';
   Address(5).State = 'MN';
EndIf;
```

Notes

In V6 a new keyword was introduced - TEMPLATE - to save on storage when we simply want to define our own data types - i.e. DS and fields that are simply going to be used as templates for other DS and fields via LIKEDS and LIKE. This is very useful when you want to be able to use a standard definition for (say) addresses to be used in all programs to help give consistency in naming etc.

We will study that in a moment or two

But first here's the the old fixed-form version of the definitions in the chart above:

```
D Address         DS                  DIM(20) QUALIFIED
D   Street1                     30a
D   Street2                     30a
D   City                        20a
D   State                        2a
D   Zip                          5a
D   ZipPlus                      4a
```

Sorting Data Structure Arrays

SORTA can be used with DS arrays

- BUT currently Ascend/Descend keywords cannot be used
  - So %Lookup cannot use the fast binary search

So there is still a role for group fields

```
Dcl-DS  ProductInfo;
  ProductData  Dim(1000) Ascend;
    Name         Char(8)       Overlay(ProductData);
    UnitPrice    Packed(7:2)   Overlay(ProductData: *Next);
    QtyInStock   Packed(9)     Overlay(ProductData: *Next);
End-DS;

Dcl-DS  ProductInfo2  Qualified  Dim(1000);
  Name         Char(8);
  UnitPrice    Packed(7:2);
  QtyInStock   Packed(9);
End-DS;

SortA UnitPrice;
Index = %Lookup(SearchFor: UnitPrice); // Fast binary search

SortA ProductInfo2(*).UnitPrice;
Index = %Lookup(SearchFor: ProductInfo2(*).UnitPrice); // Linear search
```

Notes

The introduction of DS arrays, while a huge improvement over the old MODS (Multiple Occurrence Data Structures), does not fulfill every possible need - at least as far as performance goes.

The one thing missing from the support if the ability to specify the Ascend/Descend keywords on the array, or on the individual elements of the array. As a result %Lookup is not enabled to use the fast binary search available with conventional arrays and those created with group fields.

This chart demonstrates the differences in the two methods for defining such DS and sorting and searching them.

A Better Example - A "Spreadsheet"

The DS array SalesByYear has ten elements

- One for each of 10 years of sales

Each "year" contains a 12 element array of monthly sales values

- And a field containing the total sales for that year

For each year in turn the logic builds the annual total

```
D SalesByYear     DS                  Dim(10) QUALIFIED
D   Sales4Month                  7p 2 Dim(12)
D   Total4Year                   9p 2 Inz

  For Y = 1 to %Elem(SalesByYear);
     SalesByYear(Y).Total4Year = %XFoot(SalesByYear(Y).Sales4Month);
  EndFor;

    // Free-from version

  Dcl-DS  SalesByYear  Dim(10)  QUALIFIED;
   Sales4Month  Packed(7:2)  Dim(12);
   Total4Year   Packed(9:2);
  End-DS;

  For Y = 1 to %Elem(SalesByYear);
     SalesByYear(Y).Total4Year = %XFoot(SalesByYear(Y).Sales4Month);
  EndFor;
```

Notes

This example uses %XFoot, but in some cases you will need to manually loop through the inner array to build the totals. For example if you wanted to create year-to-date values in the array, the next example demonstrates that.

Processing Multi-Dim Array V2

A similar situation

- But this time we need to build Year-To-Date sales figures

We must loop through the inner array to build the totals

- XFoot can't help us here

```
Dcl-DS  SalesStats  Dim(10)  QUALIFIED;
  Sales4Month  Packed(7:2)  Dim(12);
  SalesYearTD  Packed(9:2)  Dim(12);
End-DS;
Dcl-S Y           Int(3);
DCL-S M           Int(3);
DCL-S Total4Year  Packed(9:2); // Work field for Year-to-date total

For Y = 1 to %Elem(SalesStats);
  Total4Year = 0;  // Set YearTD to zero at start of year
  For M = 1 to %Elem(SalesStats.Sales4Month);
    Total4Year += SalesStats(Y).Sales4Month(M);  // Add current month
    SalesStats(Y).SalesYearTD(M) = Total4Year;   // And store in YearTD
  EndFor;
EndFor;
```

Notes

Similar techniques can be used to build (say) quarterly totals.

The point we're trying to make here is that RPG arrays can now do everything that an Excel spreadsheet can do and more - without resorting to obscure and hard to maintain programming techniques.

Creating "Data Types" - V6's TEMPLATE

This code creates a new "Address" data type

- But no storage is allocated to it - it is just a "note" to the compiler

MailAddr & ShipAddr are defined using this data type

- They will contain the same field definitions as Address
- Because the LIKEDS keyword is used they are implicitly QUALIFIED

Place standard definitions like this in a /COPY member

- Then you can include ll of your standard definitions with one simple line

```
      /COPY StdTypes
 2+    dcl-ds Address_T       Template;
 3+      Street1     char(30);
 4+      Street2     char(30);
 5+      City        char(30);
 6+      State       char(2);
 7+      Zip         zoned(5);
 8+      ZipPlus     zoned(4);
 9+    end-ds;

       dcl-ds MailAddr         LikeDs(Address_T);
       dcl-ds ShipAddr         LikeDs(Address_T);
```

Notes

Most modern programming languages include the ability to effectively create your own data types. V5R1 RPG gave us the QUALIFIED and LIKEDS keywords. That was a first step along the path as they enabled us to define one DS as looking exactly like another. V5R2 gave us the ability to use LIKEDS within a DS - allowing us to nest data structures one within another.

In both cases however, the definitions to use memory and the only option which avoided this (using the BASED keyword on the definition) precluded the use of initial values and could lead to errors if the programmer mistakenly referenced one of the fields in the DS. If you are unfamiliar with this technique you'll find an example on a later notes page.

With the new TEMPLATE keyword, RPG is all "growed up" and can now define templates for use by LIKE and LIKEDS that do not use memory and can be given initial values. In fact you can even define files with the TEMPLATE keyword - but more on that later.

"Data Types" - More ...

Partner400

Illegal usage of templates is detected by compiler

- e.g. If you try to store anything in them

Initial values can be specified and can be cloned via Inz(*LIKEDS)

TEMPLATE can also be used with Stand-alone fields and files

- Templates can be referenced by %SIZE, %LEN, %ELEM and %DECPOS

```
      /COPY StdTypes
2+     dcl-ds Address_T        Template;
3+       Street1     char(30);
4+       Street2     char(30);
5+       City        char(30);
6+       State       char(2);
7+       Zip         zoned(5);
8+       ZipPlus     zoned(4);
9+     end-ds;

       dcl-ds MailAddr         LikeDs(Address_T);
       dcl-ds ShipAddr         LikeDs(Address_T);

       If MailAddr.City = 'Rochester';
         MailAddr.State = 'MN';
       EndIf;
```

Notes

Partner400

You may see pre-V6 examples where templates are defined using the Based keyword. The only reason this was done was to avoid wasting static memory on structures that would never be used to hold data - only to provide a definition to be "cloned".

With V6's TEMPLATE keyword we now have an engineered solution to this problem so use it! DOn't use the old method.

Entering the Fourth Dimension

You can create as many dimensions as you like but ...

- You cannot exceed 16Mb in total for any given data item

```
Dcl-DS  Divisions  Dim(5) Qualified;
  DivCode  ZONED(2);
  Dcl-DS  Departments  Dim(10);
    DeptCode  ZONED(3);
    Dcl-DS  Products  Dim(99);
      ProdCode     ZONED(5);
      MonthsSales  PACKED(9:2)  Dim(12);
    End-DS;
  End-DS;
End-DS;


Divisions(1).Departments(1).Products(1).MonthsSales(1)
       = TotalSalesForMonth;
```

Notes - DS Size issues prior to V6

The steps below show how to calculate the total size of a nested structure. As you will see you need to start at the innermost level - in this case the Products (or technically Divisions(n).Departments(m).Products) DS.

```
(C) Dcl-DS  Divisions  Dim(5) Qualified;
      DivCode  ZONED(2);
(B)   Dcl-DS  Departments  Dim(10);
        DeptCode  ZONED(3);
(A)     Dcl-DS  Products  Dim(99);
          ProdCode     ZONED(5);
          MonthsSales  PACKED(9:2) Dim(12);
        End-DS;
      End-DS;
    End-DS;
```

(A) Each individual element in **Products** is made up of the 3 digit Department code and a 12 element array of 9,2 sales values. Each sales value is 5 Bytes long so the size of ProductData is: 5 + (12 x 5) = 65 bytes.

(B) An element of Departments consists of the 3 byte Department Code and 99 instances of the Products Array. So the size is: 3 + (99 X 65) = 6,438 bytes.

(C) At the Divisions level, each element consists of the 2 byte Division Code and 10 instances of the Departments array. So its size is: 2 + (10 X 6,438) = 64,382.

D-Spec Version of the Fourth Dimension

With the old approach you had to code individual DS for each level

- Then use LIKEDS to nest them one within the other

Notice the use of the TEMPLATE keyword

- More on this in a moment

```
D Divisions       DS                  QUALIFIED Dim(5)
D   DivCode                     2s 0
D   Departments                       LikeDS(DeptData) Dim(10)

D DeptData        DS                  QUALIFIED TEMPLATE
D   DeptCode                    3s 0
D   Products                          LikeDS(ProductData) Dim(99)

D ProductData     DS                  QUALIFIED TEMPLATE
D   ProdCode                    5s 0
D   MonthsSales                 9p 2 Dim(12)


  Divisions(1).Departments(1).Products(1).MonthsSales(1) = 0;
```

Notes

Varying Length Fields

Partner400

Defined by specifying VarChar data type

- Or adding the keyword VARYING to a type "A" fields on the D-spec

Actual storage used is length + 2

- Or length + 4 for fields > 64K long

The extra bytes hold a count of the portion of the field in use

```
D varyingStruct   DS
D  varyField                    256a   Varying Inz

 // These fields defined just to show the layout of a varying field
D   length                       5i 0 Overlay(varyField)
D   data                        256a   Overlay(varyField: *Next)
```

```
Dcl-Ds  varyingStruct;
  varyField  Varchar(256)  Inz;

  // These fields defined just to show the layout of a varying field
  length     Int(5)   Overlay(varyField);
  data       Char(256)  Overlay(varyField: *Next);
End-Ds;
```

Notes

Partner400

Varying length fields have two components: the current length that is represented by a 2-byte integer in the first two positions, followed by the actual data. Actually longer fields (i.e. > 64k in length) need a 4-byte integer but most of the ones you define will be short enough to only have 2-byte headers.

Varying length fields are differentiated from regular character fields by the use of the keyword "Varying" or by using VarChar instead of Char in free-form definitions.

You should train yourself to always code the INZ keyword to ensure that the length field is set correctly. This is critical when varying length fields are incorporated in data structures. Why? Because by default, data structures are initialized to spaces (hex 40) and that causes havoc when interpreted as the field length!

In the code example that I have redefined the two components as separate fields to demonstrate the layout.

Varying Length Fields

Partner400

Invaluable when iteratively building strings

- Orders of magnitude faster than using %TrimR and adding new content

The second version is much, much faster

- The Notes page explains why if it is not obvious to you

```
dcl-s  buffer CHAR(1000000)  Inz;
// Using a fixed length field to build the CSV buffer
DoU %EOF( CustMast );
   buffer = %TrimR(longFixed) + '"' +
            CustCode + '",' +
           %Char(BalanceDue) + CRLF;
   Read CustMast;
EndDo;
```

```
dcl-s  buffer VARCHAR(1000000)  Inz;
// Using a varying length field to build the CSV buffer
DoU %EOF( CustMast );
   buffer += '"' + CustCode + '",' +
             %Char(BalanceDue) + CRLF;
   Read CustMast;
EndDo;
```

Notes

Partner400

Whenever the content of a varying length field is changed, the compiler adjusts the length portion to reflect the new content.

Note that you should always use %Trimx when loading data from a fixed length field into a varying length field, otherwise any trailing blanks will be counted in the field length. Any time you want to know how long the field is, use the %Len() built-in function to obtain the current value.

Look at the two pieces of code on the chart. Both of them build a string of 1,000,000 bytes containing comma separated values extracted from a file. At first glance there is very little difference in the logic, but the second one can run hundreds or even thousands of times faster.

The reason is simple. The second version makes use of a varying length field to build up the result string! This difference in speed is easy to understand if you think about what is going on under the hood. The first version uses a fixed length target string so these are the steps that take place:

1: Work out where the last non-space character is (%TrimR)

2: Add the content in the next and subsequent positions.

3: If the buffer is not full, add blanks to fill it.

This process is repeated for each new value added to the string. Notice that having carefully padded the string with blanks at step 3, the very next thing we will do back at step 1 is to work out how many trailing blanks there are so that we can strip them off again!

When using the variable length buffer field it is all much simpler. We already know where the last valid character is (the length tells us this) - so all that has to happen is that we copy in the data starting in the length + 1 position and then adjust the length so that it includes the new data. Really trivial and no wasted time searching for the end of the string or adding padding blanks.

Sorting Subfiles

Want to sort subfiles?

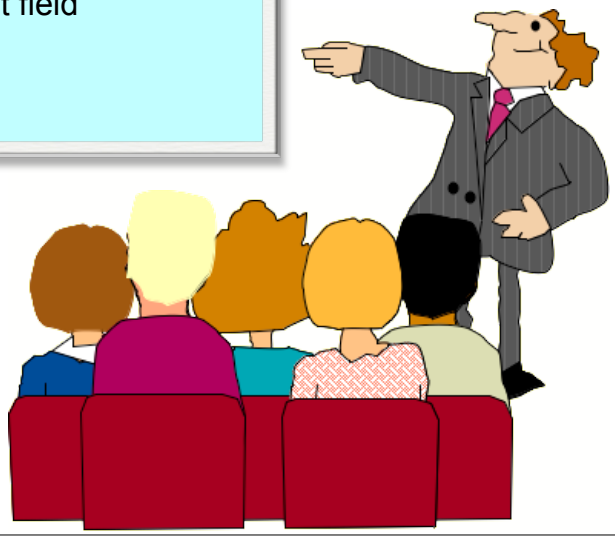
Here's an example using many new features

The LIKEREC keyword

DS Arrays

I/O Operations using the result field

Naming Indicators



Notes

Requests relating to how to sort subfiles are one of those most commonly posed on Internet lists.

This simple demonstration program highlights a very simple approach that can easily be adapted to almost any sorting requirement. The "secret" lies in loading the entire subfile into memory at the start of the process. Sorting it and then redisplaying becomes not only a trivial task but also one that performs very well.

You may need to get to release 6.1 before you can define a subfile with 9,999 records in it - but you can adapt the process to use a user space and simulate the array aspects. If you use a page-at-a-time approach to loading your subfile, you will also have to adapt the logic so that the subfile is fully loaded before you attempt to display it after a sort request - but this is not hard to do.

As you will see we are using many of the techniques we have described in this session in the program. Note that a full source listing (including the file definitions) is included at the end of this handout.

Subfile Sort - Basic Flow

1. **Load all records into the subfile array**
2. **Load the subfile from the array**
3. **Display the subfile**
4. **If sort requested determine which sequencing routine to use**
5. **Call the sort (qsort in this case)**
6. **Clear the subfile**
7. **Loop back to 2 until the user tells us we are finished**

```
(1)   Count = LoadArray();   // Store count of records loaded

      DoU (ExitRequest)
(2)     LoadSubfile(Count);  // Copy data from array into subfile
(3)     DisplaySubfile();
(4)     Select ...           // Determine sort sequence
(5)       Sort ...
(6)     ClearSubfile();      // Clear out subfile ready for reload
(7)   EndDo;
```

Notes

A common user request is to sort subfile data on the screen. Using this sorting technique, this can be fairly simply accomplished. The data is first loaded into the array, then the subfile is loaded from the array. If/when the user requests a different sequence to the data in the subfile, don't retrieve it again from the file. Instead simply re-sequence the array data using the simple SORTA technique and reload the subfile from the array.

The biggest advantage to this approach is that additional sort options can be added to a program in a matter of minutes. Add the indicator to the display file, write the sort routine, add the extra two lines to the SELECT clause and you are pretty much done.

This method is of course only good for data that does not need to be up to the minute. Since the data is in the array any changes being made to the data in the database will not be reflected. In our experience though, the user finds it much less confusing if the data is the same after a sort as they saw previously - it confuses them if they sort something into a different sequence and then can't find the record they were looking at before!

If it is vital for the data to always be current, then using embedded SQL with an appropriate ORDER BY clause is a better way to go.

1. The LoadArray Routine

The Read loads data directly into the ProductRec DS

- ProductRec itself is defined using LikeRec(PRODUCTR)

Eval-Corr then populates the subfile array element

- Which itself was defined with LikeRec(ProdSfl: *Output)

This example uses a single file and no calculations

- But as you can imagine any logic you like can be added to the load routine

```
Dcl-Proc LoadArray;
Dcl-PI   *N  Int(10);
Dcl-DS ProductRec LikeRec(PRODUCTR)
Dcl-S n  Int(10);
// Read all records and load array of subfile records
DoU %EOF(Product);
  Read ProductR ProductRec;

  If Not %Eof(Product);
    n +=1;
    Eval-Corr SubfileRec(n) = ProductRec; // Load subfile rec
  EndIf;
EndDo;
Return n; // Return record count
```

2. The LoadSubfile Routine

If this were any simpler it would have written itself!

RecordCount is passed in as a parameter

- **And used to control the For loop**

Write uses the subfile DS array element as the data source

- **With the RRN being used as the array subscript**

```
Dcl-Proc LoadSubfile;
Dcl-PI   *N;

Dcl-S RecordCount  Int(10);

For RRN = 1 to RecordCount;
  Write ProdSfl SubfileRec(RRN);
EndFor;

End-Proc;
```

4 & 5 Sort Selection Portion of Mainline

Partner400

Note use of Named Indicators for testing input F key requests

- SortDS is the name of the prototype for qsort()

```
(4)  Select;        // Determine sort option selected by user
       When SortPrCode;
(5)      qsort(SubfileRec(1):    // Pass address of first element
               Count:
               %Size(SubfileRec):
               %PAddr( SortProduct ); // Procedure for product code sort
       When SortDesc;
(5)      qsort(SubfileRec(1):
               Count:
               %Size(SubfileRec):
               %PAddr( SortDescr );
       Other;       // Default to Product sort
(5)      qsort(SubfileRec(1):
               Count:
               %Size(SubfileRec):
               %PAddr( SortProduct );
     EndSl;
```

5. The SortProduct Routine

Partner400

The only difference between the sorts is the field comparison

- Unlike SORTA - qsort() can handle multiple key sorts
  - You just write the RPG code to do it!
- We use LikeRec to define the parms
  - The qualified individual field name (e.g. Element2.ProdCd) can then be used in the comparisons

```
Dcl-Proc SortProduct;
Dcl-PI   *N  Int(10);
  Element1  LikeRec(ProdSfl: *Output)
  Element2  LikeRec(ProdSfl: *Output)

Select;
  When Element1.ProdCd > Element2.ProdCd;
    Return High;
  When Element1.ProdCd < Element2.ProdCd;
    Return Low;
  Other;
    Return Equal;
  EndSl;

End-Proc;
```

An Alternative Sort Selection Approach

SORTA can now sequence DS arrays so ...

- It could be used for simple sorts like the Product Code
- We will normally also need to use %SubArr
  - Otherwise we'll be sorting "empty" elements

You'll soon see though that there is still a place for qsort

- It can do things that SORTA can only dream about !

```
(4)   Select;        // Determine sort option selected by user
        When SortPrCode;
(5)       SORTA %SubArr(SubfileRec(*).ProdCd: 1: Count); // Sort Product Cd
        When SortDesc;
(5)       qsort(SubfileRec(1):
                Count:
                %Size(SubfileRec):
                %PAddr( SortDescr );
        Other;       // Default to Product sort
(5)       qsort(SubfileRec(1):
                Count:
                %Size(SubfileRec):
                %PAddr( SortProduct );
      EndSl;
```

A Sample Multi-Key Sort Routine

Unlike SORTA qsort() can use any logic you like for sequencing

- The most commonly used variant is to accommodate multiple keys
  - e.g. To sort City in State as in this example

```
Dcl-Proc CityInState;
Dcl-PI   *N  Int(10);
  Element1  LikeRec(CustAddress: *Output)
  Element2  LikeRec(CustAddress: *Output)

Select;
  When Element1.State > Element2.State;
    Return High;
  When Element1.State < Element2.State;
    Return Low;
  When Element1.City > Element2.City;
    Return High;
  When Element1.City < Element2.City;
    Return Low;
  Other;
    Return Equal;
  EndSl;

End-Proc;
```

Any Questions ?

Partner400

?

Please e-mail Jon at:
Jon.Paris @ Partner400.com
for any questions