Recent releases of RPG IV have taken the data definition capabilities of the language to new heights. At long last RPG has "grown up" and things that were once impossible - like a simple 2 dimensional array - are now possible.

In addition there are many capabilities that lurk in the D-spec that many RPGers have simply missed. Adding them to your tool box can make your programs simpler and more powerful.

We'll also briefly introduce you to the new data definition features that arrived with the V6.1 release of IBM i.

The authors, Jon Paris and Susan Gantner, are co-founders of Partner400, a firm specializing in customized education and mentoring services for IBM i developers. Susan and Jon are devoted to educating developers on techniques and technologies to extend and modernize their applications and development environments. This is done via on-site custom classes as well as conferences and user group events. As part of Partner400, they are also founding members of the System i Developer consortium of educators. Visit www.SystemiDeveloper.com for more information.

Together the partners author regular technical articles for the IBM publication, IBM Systems Magazine, IBM i edition (formerly iSeries Magazine and eServer Magazine, iSeries edition), and the companion electronic newsletter, IBM i EXTRA (formerly iSeries Extra).

You may view articles in current and past issues and/or subscribe to the free newsletter or the magazine on the web. You can also view Jon and Susan's blog: www.ibsystemsgram.com/IBMi

Feel free to contact the authors at: Contact @ partner400.com or visit the Partner400 web site at www.partner400.com
Agenda

A couple of reminders on basics
  Plus highlights of V5 additions
  And changes in limits in 6.1
Mapping Indicators
  Using the INDDS keyword
  Overlaying the *IN array
D-spec discoveries
  Little known features
Sort and Search
**Names on the D-spec**

**External Names**
- The system's 10 character limit still applies
  - Unless you are on V7 where use of the ALIAS name is possible
  - But only works with DS I/O
- More on mapping of file fields to Data Structures later

**Internal Names**
- Can be up to 4096 characters long
  - Yes really!
- Ellipsis (...) are used to continue names that require multiple lines
  - Not often used except for when a procedure name exceeds 14 characters
  - Or when the field name is indented by a number of characters
  - I tend to use it more for subprocedures
- When used the balance of the definition is placed on the next line

---

Will you ever use a 4,000 character name? Of course not - but the ability to extend names beyond the fixed 14 character length allowed on the D-specs is an important one.

Why? First you should always leave at least one space in front of the name - that leaves 13 characters for the name. You should always indent the names of fields in data structures - and that leaves us with only 11 or 12 characters for the name.

With this support, when a name resists attempts at a meaningful abbreviation that will fit in the 14 character box you don't have to spend ages trying to decide which bits to chop out!
**D-Spec Keywords - OVERLAY**

OVERLAY replaces the need for From/To definitions

- We'll be using this feature a lot in later examples - hence this review
- *NEXT means that the field follows the previous one OVERLAY'd
  - *NEXT could have been used in place of the "16" in the first example
- OVERLAY can be used against a field name
  - Or against the DS name

```
D personalData    DS                  INZ
D  name                         30
D  firstName                   15    Overlay(name)
D  lastName                    15    Overlay(name :16)
D  phone_No                     10
D  area_Code                    3    Overlay(phone_No)
D  local_No                     7    Overlay(phone_No :*Next)
D  status                        1    INZ('I')
D myDataStruct    DS
D  numField                      7s 0
D  charField                     7a   Overlay(myDataStruct)
```

Note the use of "floating" subfield names to show structure.

Area_Code overlays first 3 positions of phone number (pos 31-33 of DS). Local_No overlays the next 7 positions of phone number (i.e. pos 34-40 of the DS).

Use of OVERLAY to "subdivide" or redefine fields is recommended because of the increased maintainability. If fields are added to the DS, or the length of a field is changed, fields defined in this way will "self-adjust" - unlike the old From / To method that requires multiple modifications whenever such changes are made.

The INZ keyword can be used at the DS level and/or the subfield level. Note the entire DS initialized to blanks except status, initialized to "I".

Rules for OVERLAY use:

- Optional position parameter may be a numeric literal, numeric named constant, or a built-in function
- *NEXT may be used instead of the fixed position parameter.
- Length notation must be used on the overlaying field.
- If an array is overlaid, keyword applies to each element of the array.
**Qualified Data Names**

RPG's data definition support finally "grows up"

- Among other things, Data Structures can now be specified as arrays
  - Bye Bye and good riddance to Multiple Occurrence Data Structures!
  - More on this later

**DS Keyword QUALIFIED**

- References to fields must be qualified to the parent DS name
- Qualification is through the use of a dot (period) notation
  - In the DS below the fields are as `date.year`, `date.month` and `date.day`
- Keyword must be used if the DS is an array

```plaintext
D date   DS          QUALIFIED
D year   4s 0
D month  2s 0
D day    5s 0
/Free

date.year = 2008;
date.month = 11;
date.day = 4;
```

Once a DS is qualified, the same field name can appear in multiple Data Structures. Qualified data names are particularly useful when you want to /Copy some code containing standard Data Structures into a program since even if any of the subfield names happen to already exist, they will not cause an error.

Note also that you are no longer limited to Multiple Occurrence Data Structures - you can now define data Structure arrays through the DIM keyword on the DS definition. More later.
### Qualified Data Names

**DS Keyword LIKEDS**

- The 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
- Use of LIKEDS implicitly adds the QUALIFIED keyword to the new DS
  - The original DS does not have to be qualified
- Use INZ option "LIKE DS" to clone initialization values

---

<table>
<thead>
<tr>
<th>Field</th>
<th>DS Type</th>
<th>Dim</th>
<th>Qualified</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>DS</td>
<td>Dim(10)</td>
<td>Qualified</td>
</tr>
<tr>
<td>year</td>
<td></td>
<td>4s 0</td>
<td>Inz(2003)</td>
</tr>
<tr>
<td>month</td>
<td></td>
<td>2s 0</td>
<td>Inz(01)</td>
</tr>
<tr>
<td>day</td>
<td></td>
<td>5s 0</td>
<td>Inz(01)</td>
</tr>
<tr>
<td>orderDate</td>
<td>DS</td>
<td>Dim(99)</td>
<td>Qualified</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>LikeDS(date)</td>
<td>Inz(&quot;LIKE DS&quot;)</td>
</tr>
</tbody>
</table>

```
/Free
If orderDate(1).month = orderDate(2).month;
```

---

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("LIKE DS").

This means that any initial values specified in the original data structure should be replicated in the new (LIKE DS) 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("LIKE DS") to the OrderDate D spec.

There is a particularly good discussion and example of this support as it relates to passing a data a structure as a prototyped parameter in the article by Hans Boldt and Barbara Morris in IBM’s iSeries Magazine, May 2001 issue.
These new TEMPLATE keywords really lets you define your own data types, safe in the knowledge that you can’t accidentally reference the field names used in them directly.

Because no storage is allocated to templates, they work very well in a /COPY member. Imagine grouping in one or more copy files all of your shop’s standard definitions. What do addresses look like? Phone numbers, part numbers, GL code, etc. etc. You get the idea.

Templates can also include nested DS definitions, so you can create some very complex structures if you need them.

Note that although templates can’t store data, they can also be used with BIFs such as %SIZE, %LEN, %ELEM and %DECPOS to determine their characteristics.
Creating Data Types Prior to 6.1

You can effectively do the same thing at releases prior to 6.1

- But to avoid "wasting" storage you must use the BASED keyword
- But there are drawbacks
  - Some errors can only be detected at run time
  - Initialization values cannot be specified

```plaintext
D Address         DS         BASED(pTemplateOnly) QUALIFIED
D   Street1                     30a
D   Street2                     30a
D   City                        20a   VARYING
D   State                        2a
D   Zip                          5a
D   ZipPlus                      4a
D MailAddr        DS                  LikeDS(Address)
D ShipAddr        DS                  LikeDS(Address)

/Free
PrintCity = MailAddr.City + ', ' + MailAddr.State + ' ' + MailAddr.Zip;
If Address.ZipPlus <> *Blanks; // This goes "Boom" at run time ...
```

Here we have used the BASED keyword because the fields in the Address structure will never be used. The only fields we will use are those in MailAddr and ShipAddr. The pointer name pTemplateOnly is used as added documentation to warn the programmers that follow us that this is not a "real" DS - but simply a template (data type).

The keyword QUALIFIED has been used to try ensure that if we use this "data type" (it would normally have been /COPY’d in) that the field names do not conflict with fields from the data base. Since the fields can only be referenced as Address.fieldname the chances of anyone attempting to use the fields "by accident" are very small - but very small is still an issue that is not resolved until 6.1.

The main problem with using the Based keyword, keyword is twofold:
- First you cannot specify initialization values for the DS
- Second the compiler cannot detect that you did not intend to reference the fields in the DS directly - only their "clones"

If you don't like this approach, simply remove the BASED keyword - everything will work - but you need to be aware that storage will be allocated to your "template".

Note that the pointer variable pTemplateOnly will be generated by the compiler - we do not need to define it - nor would we ever set a value into it. If we have multiple "templates" we can use the same basing pointer for all of them since it will never be used to "point" to real storage.
**Nested Data Structures**

LIKEDS can also be used to nest one DS inside another

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

To reference the fields requires double qualification

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

We can also create real Data Structure arrays

<table>
<thead>
<tr>
<th>D Address</th>
<th>DS</th>
<th>TEMPLATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Street1</td>
<td>30a</td>
<td></td>
</tr>
<tr>
<td>D Street2</td>
<td>30a</td>
<td></td>
</tr>
<tr>
<td>D City</td>
<td>20a</td>
<td>VARYING</td>
</tr>
<tr>
<td>D State</td>
<td>2a</td>
<td></td>
</tr>
<tr>
<td>D Zip</td>
<td>5s</td>
<td>0</td>
</tr>
<tr>
<td>D ZipPlus</td>
<td>4s</td>
<td>0</td>
</tr>
</tbody>
</table>

D InvInfo        DS QUALIFIED DIM(20)
D MailAddr       LikeDS(Address)
D ShipAddr       LikeDS(Address)

/Free

If InvInfo(1).MailAddr.State = InvInfo(2).MailAddr.State;

In V5R2 IBM added the ability to nest one DS inside another. Not only that but we can also define arrays of Data Structures. This is a much better approach than that provided by the old Multiple Occurrence DS support. Using the MODS approach, it was impossible to directly compare the content of a field in the first element with the same field in the second. Using this support it is simple as shown in the chart above.
Multi Dimensional Array Example

The DS array SalesByYear has ten elements

- Each of which contains a 12 element array and a total field
- Both sets of logic will sum up the elements in each Sales4Month array and place the total in the corresponding Total4Year field

<table>
<thead>
<tr>
<th>D SalesByYear</th>
<th>DS</th>
<th>Dim(10) QUALIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Sales4Month</td>
<td>7p 2 Dim(12)</td>
<td></td>
</tr>
<tr>
<td>D Total4Year</td>
<td>9p 2 Inz</td>
<td></td>
</tr>
</tbody>
</table>

C For Y = 1 to %Elem(SalesByYear)
C Eval SalesByYear(Y).Total4Year = %Foot(SalesByYear(Y).Sales4Month)
C EndFor

OR

C For Y = 1 to %Elem(SalesByYear)
C For M = 1 to %Elem(SalesByYear.Sales4Month)
C Eval SalesByYear(Y).Total4Year = SalesByYear(Y).Total4Year + SalesByYear(Y).Sales4Month(M)
C EndFor
C EndFor

Note that the syntax shown in the chart for %ELEM may change in the future. Currently the compiler will handle the syntax shown correctly, but cannot handle a third level of array.

For example - if we were to extend our example by adding the DS "Sales" as shown below:

<table>
<thead>
<tr>
<th>D Sales</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>D Stuff</td>
<td>LIKEDS(SalesByYear) Dim(5)</td>
</tr>
</tbody>
</table>

the compiler will subsequently reject the syntax %elem(Sales.Stuff.Sales4Month) **

This does not affect the ability of such structures to be used. For example:

```
Sales(1).Stuff(1).Sales4Month(1) = 0;
```

will work just fine. It is only when using functions such as %Elem that the problem arises.

In order to fix the problem the syntax may need to be changed slightly so, if in future your %Elem fails to compile, this may be the reason.

** In case you are wondering, it should produce the answer 12, since that is the number of elements in the Sales4Month array.
Entering the Fourth Dimension

You can create any nested DS structure as you like but ...

- An element of the top level DS still cannot exceed the 64K DS limit
  - This changes in 6.1 where the limit rises to 16Mb!
- In the example, changing Departments to DIM(11) breaks the old limit
  - See the notes page for the size calculations
- But increasing Divisions to (say) DIM(99) has no impact

<table>
<thead>
<tr>
<th>D Divisions</th>
<th>DS</th>
<th>QUALIFIED Dim(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D DivCode</td>
<td>2s 0</td>
<td></td>
</tr>
<tr>
<td>D Departments</td>
<td>LikeDS(DeptData) Dim(10)</td>
<td></td>
</tr>
<tr>
<td>D DeptData</td>
<td>DS</td>
<td>Based(p) QUALIFIED</td>
</tr>
<tr>
<td>D DeptCode</td>
<td>3s 0</td>
<td></td>
</tr>
<tr>
<td>D Products</td>
<td>LikeDS(ProductData) Dim(99)</td>
<td></td>
</tr>
<tr>
<td>D ProductData</td>
<td>DS</td>
<td>Based(p) QUALIFIED</td>
</tr>
<tr>
<td>D ProdCode</td>
<td>5s 0</td>
<td></td>
</tr>
<tr>
<td>D MonthsSales</td>
<td>9p 2 Dim(12)</td>
<td></td>
</tr>
</tbody>
</table>

/free
Eval Divisions(1).Departments(1).Products(1).MonthsSales(1) = 0;

To see why the simple change of increasing the dimension of the Department DS array to 11 causes the size limit to study the size of the nested DSs.

<table>
<thead>
<tr>
<th>D ProductData</th>
<th>DS</th>
<th>Based(p) QUALIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>D ProdCode</td>
<td>5s 0</td>
<td></td>
</tr>
<tr>
<td>D MonthsSales</td>
<td>9p 2 Dim(12)</td>
<td></td>
</tr>
</tbody>
</table>

ProductData is made up of the 5 byte Department code and a 12 element array. Each element is 5 Bytes long so the size of ProductData is: 5 + (12 x 5) = 65 bytes.

<table>
<thead>
<tr>
<th>D DeptData</th>
<th>DS</th>
<th>Based(p) QUALIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td>D DeptCode</td>
<td>3s 0</td>
<td></td>
</tr>
<tr>
<td>D Product</td>
<td>LikeDS(ProductData) Dim(99)</td>
<td></td>
</tr>
</tbody>
</table>

DeptData consists of the 3 byte Department Code and 99 instances of the ProductData DS Array. So the size is: 3 + (99 X 65) = 6,438 bytes.

<table>
<thead>
<tr>
<th>D Division</th>
<th>DS</th>
<th>QUALIFIED Dim(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D DivCode</td>
<td>2s 0</td>
<td></td>
</tr>
<tr>
<td>D Department</td>
<td>LikeDS(DeptData) Dim(10)</td>
<td></td>
</tr>
</tbody>
</table>

Division consists of the 2 byte Division Code and 10 instances of the DeptData DS array. So its size is: 2 + (10 X 6,438) = 64,382. This is under the limit of 65,535, but not by much.

Simply increasing the DIM on the Department DS array to 11 raises the requirement to 2 + (11 X 6,438) = 70,820 and this exceeds the current compiler limit.
6.1 - Increased Size Limits

Maximum size for a field or DS is now 16Mb
- 16,773,104 if you want to be precise!
- Previous limit was 64K (65,535)

Small problem - D-spec only has 7 positions available
- Therefore limit for length definition is a mere 9,999,999!

New D-spec keyword LEN(n) deals with this
- Can be used in place of conventional length definition if you wish
- Can also be used to define the length of a DS

| D reallyBig       | S        | 9999999a   |
| D evenBigger      | S        | a LEN(16000000) |
| D tinyField       | S        | a LEN(1)   |

6.1 - Increased Size Limits - contd.

Variable length fields can now be up to 16,773,100 characters
- Previous limit was 64K (65,535)

This extra length requires a 4 byte count
- Not 2 bytes as was previously used

Compiler defaults to count of 2 or 4 based on field length
- VARYING(4) can be specified to force use of 4 byte length field

%ADDR supports extra parameter *DATA
- Supplies the address of the start of the data portion of the varying field
  - i.e. Start of field +2 or +4 depending on the size of the length portion

// This field is 1,002 bytes long
D shortVary       s 1000a Varying

// This one is 16,000,004 bytes long
D longVary        s a LEN(16000000) Varying

// And this one is 1,004 bytes
D v               s 1000a Varying(4)
6.1 - Increased Array Limits

Arrays can now have up to 16,773,104 elements
  • But only if each element is only 1 byte long!

Limit is now based on maximum size of the array (16Mb)
  • Maximum number of elements is 16Mb divided by the element size
    - So for four byte elements, the maximum would be 4,193,276
    - And 838,655 for 20 byte elements
  • Previous limit was 32,767 elements

Greatly increases the utility of Nested Data Structures
  • And therefore the utility of V5R4’s XML-INTO operations
  • The 64K limit was a real problem

While there will always be occasions when we want bigger fields and arrays than even the latest release of RPG provides, 6.1 goes a very long way to removing them.

It really is time to start looking harder at the use of arrays in our RPG programs, and these new limits should relieve many of the size issues associated with the use of XML processing in RPG programs.
Externally Described DS Extensions

EXTNAME has a new optional parameter

- It specifies which category of fields to include in the DS
  - *INPUT, *OUTPUT, *KEY or *ALL
    - The default is *INPUT which was the previous behaviour

New keyword LIKERC(recordname { : fieldcategory} )

- Defines a DS as having the same format as the named record
  - Can also be used for DS Subfield, Return value, or Prototyped parameter
    - VERY useful for Subprocedures that return a record or record set
- Field categories, and default are the same as for EXTNAME
- Field sequence matches the record buffer

Use as Result field with I/O ops to speed them up and more

<table>
<thead>
<tr>
<th>FMSTDSP</th>
<th>CF</th>
<th>E</th>
<th>Workstn</th>
</tr>
</thead>
<tbody>
<tr>
<td>D SubFileRecs</td>
<td>DS</td>
<td>LikeRec(EMPSEL:*Output) Dim(20)</td>
<td></td>
</tr>
<tr>
<td>D DisplayData</td>
<td>DS</td>
<td>Qualified</td>
<td></td>
</tr>
<tr>
<td>D SelectInp</td>
<td></td>
<td>LikeRec(Select:*input)</td>
<td></td>
</tr>
<tr>
<td>D SelectOut</td>
<td></td>
<td>LikeRec(Select:*output)</td>
<td></td>
</tr>
</tbody>
</table>

In addition to being able to specify that a DS is externally described, we can now use the keyword LIKERC to specify that the structure takes on the “shape” of a specific named record format. Of course that record format has to exist in the program (via an F spec definition of the containing file).

In what way is this different from a conventional externally described DS? Ask anyone who has ever tried to use a display file as the source of an external description. An externally described DS only contains Input fields. LIKERC goes beyond this by providing an optional second parameter that allows you to specify that the structure should contain all the fields from the record format (*ALL), or be limited to input (*INPUT), output (*OUTPUT), or key (*KEY) fields.

These new features have also been added to the “conventional” externally described DS by adding the additional parameter to the EXTNAME keyword.

This is a very useful facility for those who are attempting to "Webulate" existing RPG programs. As you will see later - the *KEY option has an additional benefit.
You can now specify a DS as the Result of an I/O operation

- Previously this could only be done for program described files
  - You must specify the Format name - until V5R3 (See the notes pages)

Remember the LIKERECC keyword?

- Use it, or an externally described DS, as the resulting DS

The resulting code may run faster - Why?

- Because the whole record buffer is moved directly into the DS
  - Field by field moves are not generated
- No data conversion is required for numeric or date fields
  - They will retain their original format

<table>
<thead>
<tr>
<th>FMSTDSP</th>
<th>CF</th>
<th>E</th>
<th>Workstn</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>LikeRec(Select:*Input)</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td>LikeRec(Select:*Output)</td>
</tr>
<tr>
<td>C</td>
<td>Write</td>
<td>Select</td>
<td>SelectOutput</td>
</tr>
<tr>
<td>C</td>
<td>Read</td>
<td>Select</td>
<td>SelectInput</td>
</tr>
</tbody>
</table>

6.1 Allows the use of Result field I/O with EXFMT!

You can now specify a DS as the result field of I/O operations. Previously, this support was only available for program-described files. (Program described files? What are those?). Combine this with the new LIKERECC support we discussed earlier and you have a very powerful and useful combination. Note: You must specify the record format name for the operation, the file name cannot be used until V5R3 - and even then there are severe limitations - see the notes on the following pages.

One benefit of performing the I/O operations directly into/from a DS is that performance of many I/O operations may improve, because the compiler will be able to move all of the record’s fields as a block, rather than have to move each field individually to the buffer. This is particularly useful when performing batch operations and will probably eliminate the performance advantage that COBOL has held over RPG for a long time when performing this type of operation.

Something for you to think about. The DS used to receive the record can be a nested DS or a DS array. This offers some interesting possibilities when working with subfile type lists, because a series of records can be loaded into different elements of the DS array. The array can then be sorted into any sequence requested by the user!! Interesting n’est pas! You'll probably want to use the C function qsort to do this, but it's not heard to learn. The RPG Redbook “Who Knew .....” contains a worked example. You can link to the Redbook from our web page at www.Partner400.com/RPGRedbook.htm.

6.1 allows the use of EXFMT with Result field I/O. The DS used must be built with the *ALL option so that all input and output fields are present. The compiler then moves the Output and Both fields to the output buffer and when the operator responds, copies the relevant input and both fields to the DS. This option makes it much easier to think about replacing the EXFMT with an API call later if "webulating" the program.
Agenda

A couple of reminders on basics
   Plus highlights of V5 additions
   And changes in limits in 6.1

Mapping Indicators
   Using the INDDS keyword
   Overlaying the *IN array

D-spec discoveries
   Little known features
   Sort and Search
Named Indicators and INDDS

Use real names for your display indicators!

- F-spec keyword INDDS ties the external indicators to the named DS
  - An alternative approach to the indicator definition is shown on the next page

<table>
<thead>
<tr>
<th>FDisplay</th>
<th>CF</th>
<th>E</th>
<th>WORKSTN</th>
<th>INDDS(DspInd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>DspInd</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* Response indicators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Exit</td>
<td>3</td>
<td>3N</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Return</td>
<td>12</td>
<td>12N</td>
<td></td>
</tr>
<tr>
<td>* Conditioning indicators</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Error</td>
<td>31</td>
<td>31N</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>StDateErr</td>
<td>32</td>
<td>32N</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>EndDateErr</td>
<td>33</td>
<td>33N</td>
<td></td>
</tr>
</tbody>
</table>

C If StartDate < Today
C Eval StDateErr = *On
C EndIf
C Eval EndDateErr = EndDate < StartDate
C Eval Error = StDateErr or EndDateErr
C ExFmt TestRec
C If Exit or Return

This example illustrates the use of the INDDS (indicator Data Structure).

Note the definition of the DSPIND data structure. When this is specified on the F spec, the programmer MUST use the indicators in the data structure and NOT the numbered indicators to control this file. For example, in this program, if the programmer turned on indicator *IN31 directly in the program logic, it would have zero impact on the display file. The program logic must refer to the indicator as “Error” to turn it on or off in order for it to have an impact on the display file.

This is the DDS for the display file used in the RPG example:

```
INDARA
R R TESTREC
  CF03(03) CF12(12)
  5 11 'Start Date: . . .'
  STARTDATE L 5 27DATFMT(*USA)
  32 DSPATR(RI PC)
  32 5 39 'Date cannot be earlier than today'
  7 11 'End Date: . . .'
  ENDDATE L 7 27DATFMT(*USA)
  33 DSPATR(RI PC)
  33 7 39 'Date must be later than Start Date'
  31 10 21 'Please correct above error'
```

In the example on the following page, we have also incorporated the associated indicator number in the indicator’s name. Some people like this approach as it enables the programmer to see the indicator number without having to look at the Indicator Data Structure.
Named Indicators and INDDS

Use the OVERLAY keyword to avoid From/To notation

- Some people find this a more better approach
  - Note: In this case we have incorporated the indicator number in the name

<table>
<thead>
<tr>
<th>FDisplay</th>
<th>CF</th>
<th>E</th>
<th>WORKSTN INDDS(DspInd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D DspInd</td>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* Response indicators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D Exit_3</td>
<td>N</td>
<td>Overlay(DspInd: 3)</td>
<td></td>
</tr>
<tr>
<td>D Return_12</td>
<td>N</td>
<td>Overlay(DspInd: 12)</td>
<td></td>
</tr>
<tr>
<td>* Conditioning indicators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D Error_31</td>
<td>N</td>
<td>Overlay(DspInd: 31)</td>
<td></td>
</tr>
<tr>
<td>D StDateErr_32</td>
<td>N</td>
<td>Overlay(DspInd: 32)</td>
<td></td>
</tr>
<tr>
<td>D EndDateErr_33</td>
<td>N</td>
<td>Overlay(DspInd: 33)</td>
<td></td>
</tr>
</tbody>
</table>

/Free

If StartDate < Today;
  StDateErr_32 = *On;
EndIf;

EndDateErr_33 = EndDate < StartDate;
Error_31 = StDateErr_32 or EndDateErr_33;
ExFmt TestRec;

If Exit_3 or Return_12;

Using the OVERLAY keyword achieves two things:

- First it removes the need to use the old fashioned From/To notation
- Second it makes the indicator number a little more obvious

Yes - it is a bit more typing - but you only type once - you may have to read it hundreds of times! So get over it!
**Mapping the *IN Array**

Don't want to use the INDDS approach?
- You can always map the standard indicator array with a "Based" DS

Do you use the *INKx Indicators?*
- A similar approach can be used
  - Use %Addr(*INKA) to initialize the basing pointer
  - Don't forget that there are only 24 indicators in the set!
    - There is no *INKO
  - See the Notes page for an example of how this is done

```
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    StDateErr_32                 N   Overlay(DspInd: 32)
D    EndDateErr_33                N   Overlay(DspInd: 33)
D pIndicators     S               *   Inz(%Addr(*In))
```

Unlike the INDDS approach, these named indicators DO directly affect the content of their corresponding *IN indicator. If we EVAL 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.

```
D FunctionKeys       DS                  Based(pKxIndicators)
D   thisIsKC                      N   Overlay(FunctionKeys: 3)
D   thisISKL                      N   Overlay(FunctionKeys: 12)
D pKxIndicators   S               *   Inz(%Addr(*InKA))
```
An Easy Way to Name Indicators

This is a technique can be used to gradually name indicators

- It can be readily applied during maintenance
  - Whereas the previous methods require a greater effort

It takes advantage of the fact that *IN is an array

- Constants can be used to supply the indicator number

This example uses an old technique that many folks used even in RPG/400. Simply use named constants as indexes into the *IN array. This may be a better solution for modifying existing code, since it can be applied gradually.

As you discover the purpose of an indicator, give it a name and establish a constant for the indicator value. There now - that didn't hurt did it!

Always remember - the next person who needs to know what *IN57 = *On means may be YOU!
Agenda

- A couple of reminders on basics
  - Plus highlights of V5 additions
  - And changes in limits in 6.1
- Mapping Indicators
  - Using the INDDS keyword
  - Overlaying the *IN array
- D-spec discoveries
  - Little known features
- Sort and Search

D-Spec Discoveries

- Some lesser known D-Spec talents
  - No-length fields
  - No-name fields
  - Incorporating external fields into a DS
- Group fields
  - A useful capability
**Unnamed Fields and Overlaying a DS**

**DS subfields do not need to be named!**
- But they can still have INZ values!

**Originally, the Overlay keyword only applied to subfields**
- Now, you can overlay the DS itself

**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

<table>
<thead>
<tr>
<th>D</th>
<th>Messages</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td>20a Inz('Invalid Item Code')</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>20a Inz('Too many selections')</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>20a Inz('Item Code required')</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>20a Inz('Huh?')</td>
</tr>
<tr>
<td>D</td>
<td>Msg</td>
<td>20a Overlay(Messages) Dim(4)</td>
</tr>
</tbody>
</table>

Our passionate dislike of compile-time data led us to use this technique as an alternative approach. It certainly makes life much simpler when wanting to see what values are used to initialize the array.
No-Length Subfields

Want to access the fields in a record as an array?

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

No length definition is required

- The compiler will use the length supplied by the database

Next overlay the array against the DS name

- Redefining the fields as an array
- Notice the use of the LIKE keyword

The inspiration for this example comes from one of the most commonly asked questions on RPG programming lists: “How do I directly load fields from a database record into an array?” The question normally arises when each database record contains a series of related values - for example, monthly sales figures.

The DDS for the physical file is shown. One solution is depicted here. We'll look at a 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. Instead of allocating storage for the file’s fields arbitrarily, the compiler is told to explicitly place the data in the DS. Because the DS no longer contains the Customer and Division data, we can use the simple form of the Overlay keyword. Otherwise, if we had used an externally described DS, we would have needed to place a starting position on the Overlay keyword. The code for that alternative solution is shown below. Unlike the externally described DS, The example on this chart allows you to see exactly which fields form the DS. In addition, the fields in the DS can come from multiple files.

```plaintext
D SalesData       E DS                  ExtName(Test File)
// Define array over Quarterly Sales figures
D SalesForQtr     Overlay(SalesData: 7)
D                   Like(Q1) Dim(4)
```
**Group Fields**

Sometimes it is convenient to reference a group of fields

- You might wish 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

- The compiler derives the length of the group field from the combined lengths of the subfields that OVERLAY it.

```plaintext
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)
```

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 QuarterData, 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 . . . .
SALESDATA        DS(34)                     10D . . . .
SALESFORQTR(4)   S(7,2)                     19D . . . .
```
Using SORTA with Group Fields

Want to sort an array on different keys?
- Group fields can provide an answer

ProductData is a group field
- It comprises the fields Name, UnitPrice and QtyInStock

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

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, you can only sort ascending OR descending sequence on any given array.

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                       Dim(1000) Acscend
D   ProductData                       Dim(1000)
D   Name                       8    Overlay(ProductData)
D   UnitPrice                  8    Overlay(ProductData: *Next)
D   QtyInStock                 9p 0 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                 8    Overlay(ProductDataD: *Next)
D   QtyInStock                 9p 0 Overlay(ProductDataD: *Next)
D pProductInfo    S               *   Inz(%Addr(ProductInfo))
```
Data Structures, F-Specs and Prefix

The PREFIX keyword is more flexible than you may think

- As shown below it can be used to qualify the fields in the file
- Much easier to remember which file they come from
  - Use the replace as shown below to remove an existing prefix
  - Here we are removing the first two characters

Fields will be loaded to/from the DS on each I/O operation

- This approach can be combined with the array mapping techniques discussed earlier

6.1 Allows the use of the QUALIFIED keyword on the F-spec

- All I/O must be done using a DS Result field

```
FCustomer IF E K DISK Prefix('CustRec.'): 2
D CustRec DS LikeRec(Custom01) Qualified
```

Note that the PREFIX keyword can also be used on externally described data structures to rename fields just as it does on the F spec. Originally this was useful if the programmer needed to save the values of a record to compare them to the next record's values, for example. However, the V5 capabilities to read into (or write from) a DS are a far simpler approach to the problem.

If the prefix value is enclosed in quotes, it can include a "." (period) - this allows a qualified name to be created. This same technique can be used on F-specs. In the example shown, if the record format named CUSTOM01 contained a field named CRBALANCE, then that field would be renamed as CUSTREC.BALANCE.

This feature can still be used in 6.1 but as an alternative the QUALIFIED keyword can now be coded on the F-spec.
Want to sort subfiles? Here's an example using many of the topics from this session:

- LIKEREC keyword
- DS Arrays
- I/O Operations using the result field
- Naming Indicators

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

Load all records into the subfile array

Display the subfile

If sort requested determine which sequencing routine to use

Call the sort (qsort in this case)

Clear the subfile

Loop back if we are not finished

Count = LoadArray();
DoU *In(Exit)
   LoadSubfile(Count);
   DisplaySubfile();
   Select ...
   Sort ...
   ClearSubfile();
EndDo;

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.
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)

D LoadArray PI 10I 0
D ProductRec DS LikeRec(PRODUCTR)
D n S 10I 0
/Free
// Read all records and fill 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 count of number of records

Sort Selection Portion of Mainline

Note use of *In( ConstantName ) for testing input indicators

SortDS is the name of the prototype for qsort()
- SortRoutine is a procedure pointer set by the SELECT logic
  - Allows us to code a single call to the sort - not one for each type of sort
  - Makes the code "cleaner" when multiple sort options exist

Select;  // Determine sort option
    When *In(SortPrCode);
        SortRoutine = SortProduct;
    When *In(SortDesc);
        SortRoutine = SortDescr;
    Other;  // Default to Product sort
        SortRoutine = SortProduct;
EndSl;

// Call the sort routine passing address of
// first element of subfile array
SortDS(SubfileRec(1):
    Count:
    %Size(SubfileRec):
    SortRoutine);
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

The Write specifies a subfile array element as the result field
- With the RRN being used as the array subscript
  - RRN is a global variable

```rpg
D LoadSubfile PI
D RecordCount 10I 0
/Free
For RRN = 1 to RecordCount;
  Write ProdSfl SubfileRec(RRN);
EndFor;
/End-Free
```

The ProductSort Routine

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!

Note the use of LikeRec to define the parms
- The qualified field name (e.g. Element2.ProdCd) can then be used

```rpg
D ProductSort PI 10I 0
D Element1 LikeRec(ProdSfl: *Output)
D Element2 LikeRec(ProdSfl: *Output)
/Free
Select;
  When Element1.ProdCd > Element2.ProdCd;
    Return High;
  When Element1.ProdCd < Element2.ProdCd;
    Return Low;
  Other;
    Return Equal;
EndSl;
```
### Other D Spec Keywords

<table>
<thead>
<tr>
<th>Keywords</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>Align Integer &amp; Float on natural boundary</td>
</tr>
<tr>
<td>BASED(ptr)</td>
<td>The basing pointer (ptr) contains the location of the storage holding the value for the field</td>
</tr>
<tr>
<td>DATFMT(format)</td>
<td>Specifies the date type for a D field</td>
</tr>
<tr>
<td>EXPORT</td>
<td>The storage for the field is allocated in this module, but may be used in other modules in this program</td>
</tr>
<tr>
<td>EXTPGM(name)</td>
<td>The program defined by this prototype</td>
</tr>
<tr>
<td>EXTPROC(name)</td>
<td>The procedure defined by this prototype</td>
</tr>
<tr>
<td>IMPORT</td>
<td>Storage for this item is allocated in another module.</td>
</tr>
<tr>
<td>OPTIONS(parm_passing_options)</td>
<td>Used in prototypes to determine how parameter is passed</td>
</tr>
<tr>
<td>PACKEVEN</td>
<td>Zero out the &quot;unused&quot; digit of an even packed field</td>
</tr>
<tr>
<td>PROCPTTR</td>
<td>Defines a pointer as a procedure pointer. Only allowed with data type ‘*’ (pointer)</td>
</tr>
<tr>
<td>STATIC</td>
<td>The data item is allocated in static storage so that it will retain it's value between procedure calls</td>
</tr>
<tr>
<td>TIMFMT</td>
<td>Specifies the time type for a T field</td>
</tr>
<tr>
<td>Keyword</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>INZ({constant{:*ExtDft or *User}})</td>
<td>Initialize to a value</td>
</tr>
<tr>
<td>OCCURS(number)</td>
<td>Number of occurrences in multiple occurrence DS</td>
</tr>
<tr>
<td>OVERLAY(name:{pos or *Next})</td>
<td>Redefines subfields in a DS</td>
</tr>
<tr>
<td>EXTNAME(name:{fmt_name}{:fieldtype} )</td>
<td>Data Structure is externally described</td>
</tr>
<tr>
<td>EXTFLD(fidname)</td>
<td>External field name being renamed</td>
</tr>
<tr>
<td>PREFIX(prefix_name)</td>
<td>Prefix subfields in an externally described DS</td>
</tr>
<tr>
<td>QUALIFIED</td>
<td>Subfields in DS are qualified by DS name</td>
</tr>
<tr>
<td>LIKEDS(DS_name)</td>
<td>Creates identical subfields as in DS_name</td>
</tr>
<tr>
<td>LIKERECC( fmt_name { : fieldtype} )</td>
<td>Similar to EXTNAME but can be used within a DS</td>
</tr>
<tr>
<td>DIM(number)</td>
<td>Creates a Data Structure Array</td>
</tr>
</tbody>
</table>

Note that the PREFIX keyword can be used on externally described data structures to rename fields as it does on the F spec. This can be quite useful if you want to create (say) before and after images of a record for example.

If the prefix value is enclosed in quotes, it can include a "." (period) - this allows a qualified name to be created.

The "EXTDFT parameter is only allowed for externally described data structures. It initializes externally described data-structure subfields with the default values from the DFT keyword in the DDS. You can override the value specified in the DDS by coding INZ with or without a parameter on the subfield specification.

The LIKERECC keyword differs from EXTNAME in one significant aspect - it can only reference a record format used in the program.