Modernizing Your File I/O with Data Structures

Brian May
IBM i Modernization Specialist
Profound Logic Software

Webmaster and Coordinator
Young i Professionals

Overview

- Discuss advantages of using data structures for I/O operations
- Review the I/O opcodes that support data structures
- Discuss the LIKEREC keyword and its abilities
- Discuss key data structures and how they are used
- Talk about local file support for subprocedures
- Discuss i 7.1 enhancement to finally allow use of long alias names in RPG

Why Use DS for I/O

What's in it for me?

Avoid Some Common "Gotchas"

- Bad data in files causing Decimal Data Errors
  - When reading in data from a file without data structures, each field is set its value individually
    - This all happens under the covers
    - If there is corrupt data in your file, the program will receive errors such as decimal data errors before your program regains control to handle them
    - Searching a table for corrupt data can be a real time sink if you don't know exactly what happened

Avoid Some Common "Gotchas"

- Bad data in tables causing Decimal Data Errors
  - When reading data from a file with data structures, the entire record is moved as a single entity
    - This prevents the run time from "touching" each field individually on input
    - Your program can then monitor for possible bad data in fields your program uses and handle them
    - If you can catch these errors, you can easily log which row in what table caused your problem so that you can review and fix it

Avoid Some Common "Gotchas"

- Name collision
  - When more than one table has columns with the same name, a couple of different problems can arise.
    - If the columns have different definitions, you can't even compile the program unless you rename them using PREFIX or some other method
    - If the columns have the same definition, your program will compile normally, but you must be diligent to keep up with your data and make sure you don't accidentally replace data unintentionally

Avoid Some Common "Gotchas"

- Name collision
  - By using qualified data structures to receive your input, names are now unique
  - Since the different formats for the same column name are now in separate qualified data structures, they are unique and your program will compile easily
  - By holding data in qualified data structures, there is no risk of overwriting any data in columns with the same name

Performance Enhancements

- For I/O intensive programs, using data structures can improve performance
  - Moving one large piece of data into a data structure is much faster than moving each individual column, especially for very long record formats
  - Storage of commonly retrieved records in a data structure array can improve performance in long running programs

Qualified Data Structures

You are really going to need these

What is a QUALIFIED DS?

- Qualified data structures instruct the compiler to require the data structure name whenever a subfield is referenced
  - Example: Customer.Email
- This qualified data structures can have subfields with the same name as other fields without name collision
- Qualified data structures are required when using certain data structure features, such as data structure arrays

How to Create a Qualified DS

- QUALIFIED keyword
  - Simplest way to make a qualified data structure

```
dCustomer          ds                  Qualified
d First_Name                     15
d Last_Name                      15
d Email                          50
d Phone                          15
```

How to Create a Qualified DS

- QUALIFIED keyword
  - Simplest way to make a qualified data structure

```
Dcl-Ds Customer Qualified ;
  First_Name Char(15) ;
  Last_Name Char(15) ;
  Email Char(50) ;
  Phone Char(15) ;
End-Ds;
```

How to Create a Qualified DS

- LIKEDS keyword
  - LIKEDS creates a data structure with the same subfields as another data structure
  - A data structure defined with LIKEDS is automatically created as qualified

```
dCustomer          ds
d First_Name                    15
d Last_Name                     15
d Email                         50
d Phone                         15

dNew_Customer      ds                    LikeDS(Customer)
```

How to Create a Qualified DS

- LIKEDS keyword
  - LIKEDS creates a data structure with the same subfields as another data structure
  - A data structure defined with LIKEDS is automatically created as qualified

```
Dcl-Ds Customer ;
  First_Name Char(15) ;
  Last_Name Char(15) ;
  Email Char(50) ;
  Phone Char(15) ;
End-Ds;

Dcl-Ds New_Customer LikeDS(Customer) ;
```

How to Create a Qualified DS

- LIKEREC keyword
  - LIKEREC creates a datastructure with subfields matching the column names and formats for a specified record format
  - A data structure defined with LIKEDS is automatically created as qualified

```
fItemMast  if   e           k Disk

dItem_Data       ds                  LikeRec(ItmMastRec)
```

How to Create a Qualified DS

- LIKEREC keyword
  - LIKEREC creates a datastructure with subfields matching the column names and formats for a specified record format
  - A data structure defined with LIKEDS is automatically created as qualified

```
Dcl-F ItemMast Keyed ;

Dcl-Ds Item_Data LikeRec(ItemMastRec) ;
```

Using Qualified DSs

- All references to subfields must have the data structure name, a period, and then the subfield name.

```
New_Customer.First_Name = 'Brian' ;
New_Customer.Last_Name = 'May' ;
New_Customer.Email = 'brian@youngiprofessionals.com' ;
New_Customer.Phone = '6625551212' ;
```

Using Qualified DSs

- EVAL-CORR operation code
  - Sets subfields with the same name and compatible data types equal
  - Saves monotonous code to set each subfield equal to its corresponding field in the other data structure

```
dCustomer         ds
d First_Name                    15
d Last_Name                     15
d Email                         50
d Phone                         15

dProspect         ds
d First_Name                    25
d Last_Name                     25
d Phone                         10
d Address1                      25
d Address2                      25
d City                          15
d State                          2
d Email                         30

 /Free

   Eval-Corr Customer = Prospect ;
```

Using Qualified DSs

- EVAL-CORR operation code
  - Sets subfields with the same name and compatible data types equal
  - Saves monotonous code to set each subfield equal to its corresponding field in the other data structure

```
Dcl-Ds Customer ;
  First_Name Char(15) ;
  Last_Name Char(15) ;
  Email Char(50) ;
  Phone Char(15) ;
End-Ds;

Dcl-Ds Prospect Qualified ;
  First_Name Char(25) ;
  Last_Name Char(25) ;
  Phone Char(10) ;
  Address1 Char(25) ;
  Address2 Char(25) ;
  City Char(15) ;
  State Char(2) ;
  Email Char(30) ;
End-Ds;

Eval-Corr Customer = Prospect ;
```

Basics of DSs for I/O

How do I make this stuff work?

Creating Data Structures

- The first thing you need when using data structures for I/O is obviously some data structures
  - Data structures can be defined using EXTNAME or LIKEREC on the data structure definition specification
  - LIKEREC creates a qualified data structure
  - EXTNAME does not create a qualified data structure unless you specify the QUALIFIED keyword
  - All input operations (READ, CHAIN, READE, etc) require you to define your data structure as *INPUT
  - The WRITE operation requires a data structure of type *OUTPUT
  - The UPDATE operation can use either type of data structure

Creating Data Structures

- Even though UPDATE can be done with an *INPUT data structure, I always use *OUTPUT
  - I like the consistency
  - I can easily check if my *INPUT and *OUTPUT data structures are different before actually performing an UPDATE

```
fMyFile    uf a e           k Disk

dMyFile_In        ds                  LikeRec(MyFileRec:*INPUT)
dMyFile_Out       ds                  LikeRec(MyFileRec:*OUTPUT)
```

Creating Data Structures

- Even though UPDATE can be done with an *INPUT data structure, I always use *OUTPUT
  - I like the consistency
  - I can easily check if my *INPUT and *OUTPUT data structures are different before actually performing an UPDATE

```
Dcl-F MyFile Usage(*Delete) Keyed ;

Dcl-Ds MyFile_In LikeRec(MyFileRec:*Input) ;
Dcl-Ds MyFile_Out LikeRec(MyFileRec:*Output) ;
```

Creating Data Structures

- Another type of data structure that you may want to have is a *KEY data structure
  - These data structures only contain the key fields
  - Subfields are ordered the same as the key
  - Easily loaded using EVAL-CORR

```
dMyFile_Key       ds                  LikeRec(MyFileRec:*KEY)
```

Creating Data Structures

- Another type of data structure that you may want to have is a *KEY data structure
  - These data structures only contain the key fields
  - Subfields are ordered the same as the key
  - Easily loaded using EVAL-CORR

```
Dcl-Ds MyFile_Key LikeRec(MyFileRec:*Key) ;
```

Retrieving Data

- Data structures can be used with any of the file input opcodes
  - READ
  - READE
  - READP
  - READPE
  - CHAIN
  - READC
- All of these operations accept a data structure in the Result Field position (in free format, they are the last parameter)

Data Retrieval Examples

```
fMyFile    uf a e            k Disk

dMyFile_In        ds                  LikeRec(MyFileRec:*INPUT)
dMyFile_Out       ds                  LikeRec(MyFileRec:*OUTPUT)

dMyFile_Key       ds                  LikeRec(MyFileRec:*KEY)

 /Free

  Read MyFile MyFile_In ;
  Chain %KDS(MyFile_Key) MyFile MyFile_In ;
  Reade %KDS(MyFile_Key) MyFile MyFile_In ;
```

Data Retrieval Examples

```
Dcl-F MyFile Usage(*Delete) Keyed ;

Dcl-Ds MyFile_In LikeRec(MyFileRec:*Input) ;
Dcl-Ds MyFile_Out LikeRec(MyFileRec:*Output) ;


Dcl-Ds MyFile_Key LikeRec(MyFileRec:*Key) ;

Read MyFile MyFile_In ;
Chain %KDS(MyFile_Key) MyFile MyFile_In ;
Reade %KDS(MyFile_Key) MyFile MyFile_In ;
```

Outputting Data

- Data structures can be used with either of the file output opcodes
  - WRITE
  - UPDATE
- Both of these operations accept a data structure in the Result Field position (in free format, they are the last parameter)

Data Output Example

```
fMyFile     uf a e             k Disk

dMyFile_In          ds                    LikeRec(MyFileRec:*INPUT)
dMyFile_Out         ds                    LikeRec(MyFileRec:*OUTPUT)

dMyFile_Key         ds                    LikeRec(MyFileRec:*KEY)

 /Free

  Chain (Key1, Key2) MyFile MyFile_In ;
  If %Found(MyFile) ;
    MyFile_Out = MyFile_In ;
    //Do Stuff

    If MyFile_Out <> MyFile_In ;
      Update MyFileRec MyFile_Out ;
    EndIf;
  EndIf;
```

Data Output Example

```
Dcl-F MyFile Usage(*Delete) Keyed ;

Dcl-Ds MyFile_In LikeRec(MyFileRec:*Input) ;
Dcl-Ds MyFile_Out LikeRec(MyFileRec:*Output) ;


Dcl-Ds MyFile_Key LikeRec(MyFileRec:*Key) ;

Chain (Key1, Key2) MyFile MyFile_In ;
If %Found(MyFile) ;
  MyFile_Out = MyFile_In ;
  // Do Stuff

  If MyFile_Out <> MyFile_In ;
    Update MyFileRec MyFile_Out ;
  EndIf ;
EndIf ;
```

Using %KDS

- %KDS() lets your data retrieval operation code know to treat the data structure like a key list
- Can be used with data structures defined with EXTNAME or LIKEREC with the *KEY option
- Can also be used with a data structure defined within the program
- Can be used even when not using data structures to receive input
- More flexible than a KLIST
  - No need to come out of free form to define it
  - Only the types of the key fields must match
  - Key fields with different lengths will be adjusted to fit automatically
  - You can specify how many key fields to use instead of creating new key lists

Putting It Together

Hold on to your hats!

Putting It All Together

- Let's say we have a header and detail table used for purchase orders
  - Order Header Table (ORDERHDR) with columns for
    - Division
    - OrderNum
    - OrderDate
    - VendorName
    - Keyed by Division and OrderNum

Putting It All Together

- Order Detail Table (ORDERDTL) with columns for
  - Division
  - OrderNum
  - Line
  - ItemNum
  - Price
  - Quantity
  - Keyed by Division, OrderNum, and Line

Putting It All Together

- For a very simple example, lets say we need a program to go through an order and update the prices for each order line
  - Assume there is an existing subprocedure called Get_Price that will handle the price look up
  - To keep journals clean, the program should only update the detail line if there is a change in the detail record

Putting It All Together

```
 //Tables
fOrderHdr  if   e           k Disk
fOrderDtl  uf   e           k Disk

 //Data Structures
dHeader_In         ds                  LikeRec(OrdHdrRec:*Input)
dHeader_Key        ds                  LikeRec(OrdHdrRec:*Key)
dDetail_In         ds                  LikeRec(OrdDtlRec:*Input)
dDetail_Out        ds                  LikeRec(OrdDtlRec:*Output)
dDetail_Key        ds                  LikeRec(OrdDtlRec:*Key)

 //Prototype for Procedure
dGet_Price         pr                  Like(Price)
d  Item                                Like(ItemNum)

 //Entry Parms
dUpdOrdPrc         pr                  ExtPgm('UPDORDPRC')
d  Parm_Division                       Like(Division)
d  Parm_OrderNum                       Like(OrderNum)

dUpdOrdPrc         pi
d  Parm_Division                       Like(Division)
d  Parm_OrderNum                       Like(OrderNum)
```

Putting It All Together

```
/Free

 Header_Key.Division = Parm_Division ;
 Header_Key.OrderNum = Parm_OrderNum ;

 Chain %KDS(Header_Key) OrderHdr Header_In ;
 If %Found(OrderHdr) ;

   Eval-Corr Detail_Key = Header_In ;
   Setll %KDS(Detail_Key:2) OrderDtl ;
   Reade %KDS(Detail_Key:2) OrderDtl Detail_In ;
   Dow Not %EOF(OrderDtl) ;
     Detail_Out = Detail_In ;
     Detail_Out.Price = Get_Price(Detail_Out.ItemNum) ;

     If Detail_Out <> Detail_In ;
       Update OrdDtlRec Detail_Out ;
     EndIf;

     Reade %KDS(Detail_Key:2) OrderDtl Detail_In ;
   EndDo;

 EndIf;

 *InLR = *On ;
 Return ;

/End-Free
```

Putting It All Together

```
 //Order Header
Dcl-F OrderHdr Keyed ;

Dcl-Ds Header_In LikeRec(OrdHdrRec:*Input) ;
Dcl-Ds Header_Key LikeRec(OrdHdrRec:*Key) ;

 //Order Detail
Dcl-F OrderDtl Usage(*Update) Keyed ;

Dcl-Ds Detail_In LikeRec(OrdDtlRec:*Input) ;
Dcl-Ds Detail_Out LikeRec(OrdDtlRec:*Output) ;
Dcl-Ds Detail_Key LikeRec(OrdDtlRec:*Key) ;

 // External Procedure
Dcl-Pr Get_Price Like(Price) ;
  Item Like(ItemNum) ;
End-Pr ;

 // Entry Parms
Dcl-Pi UpdOrdPrc ;
  Parm_Division Like(Division) ;
  Parm_OrderNum Like(OrderNum) ;
End-Pi ;
```

Walking Through the Example

- Naturally, we need to define our tables

```
 //Tables
fOrderHdr  if   e           k Disk
fOrderDtl  uf   e           k Disk
```

- Then we will need our data structures
  - An *INPUT and *KEY for the header
  - An *INPUT, *OUTPUT, and *KEY for the detail

```
 //Data Structures
dHeader_In          ds                  LikeRec(OrdHdrRec:*Input)
dHeader_Key         ds                  LikeRec(OrdHdrRec:*Key)
dDetail_In          ds                  LikeRec(OrdDtlRec:*Input)
dDetail_Out         ds                  LikeRec(OrdDtlRec:*Output)
dDetail_Key         ds                  LikeRec(OrdDtlRec:*Key)
```

Walking Through the Example

- We need to prototype our Get_Price routine
- We also need parameters for this program

```
 //Prototype for Procedure
dGet_Price        pr                  Like(Price)
d  Item                               Like(ItemNum)

 //Entry Parms
dUpdOrdPrc        pr                  ExtPgm('UPDORDPRC')
d  Parm_Division                      Like(Division)
d  Parm_OrderNum                      Like(OrderNum)

dUpdOrdPrc        pi
d  Parm_Division                      Like(Division)
d  Parm_OrderNum                      Like(OrderNum)
```

Walking Through the Example

- Now, let's load our key data structure and retrieve our order header

```
/Free

 Header_Key.Division = Parm_Division ;
 Header_Key.OrderNum = Parm_OrderNum ;

 Chain %KDS(Header_Key) OrderHdr Header_In ;
 If %Found(OrderHdr) ;
```

Walking Through the Example

- Since the column names are the same in both tables, the program uses EVAL-CORR to load the detail key with the header values
- Then a basic SETLL/READE/DOW combination to loop through the detail records

```
Eval-Corr Detail_Key = Header_In ;
Setll %KDS(Detail_Key:2) OrderDtl ;
Reade %KDS(Detail_Key:2) OrderDtl Detail_In ;
Dow Not %EOF(OrderDtl) ;
```

Walking Through the Example

- The program moves the data from the *INPUT data structure to the *OUTPUT data structure
- The program then calls the Get_Price subprocedure to populate the PRICE column

```
Detail_Out = Detail_In ;
Detail_Out.Price = Get_Price(Detail_Out.ItemNum) ;
```

Walking Through the Example

- Compare the *INPUT and *OUTPUT data structures to determine if the price changed
- If so, update the record using the *OUTPUT data structure

```
If Detail_Out <> Detail_In ;
  Update OrdDtlRec Detail_Out ;
EndIf;
```

Walking Through the Example

- Read in the next row and continue the loop
- Close out our If %Found block
- End the program

```
      Reade %KDS(Detail_Key:2) OrderDtl Detail_In ;
    EndDo;

  EndIf;

  *InLR = *On ;
  Return ;

 /End-Free
```

Newer Stuff

Some of the new features of RPG REQUIRE you to use data structures for your I/O

Local Files in Subprocedures

- One of the biggest additions to RPG in i6.1
- “F” specs are now allowed inside of subprocedures
- This will allow your subprocedures to have their own individual cursor for a file
- Because “I” and “O” specs are not allowed in subprocedures, all I/O operations must be done using data structures
- Since all local storage is automatic by default, your file will be closed when the subprocedure ends
- If you want the file to remain open after returning from the subprocedure, the STATIC keyword is allowed on the file specification

Qualified Record Formats

- As of i 6.1, the QUALIFIED keyword is allowed on file specifications
- This requires that all references to record format names must be qualified by the file name
- This is an alternative to using the RENAME keyword when you have more than one file with the same record format name
- If a file is QUALIFIED, no “I” or “O” specs are generated by the compiler
- All I/O operations on qualified files must be done using data structures

Long Column Names

- DDS, through the ALIAS keyword, and SQL have both had the ability to have long column names for years
- Until i 7.1, these more descriptive names were not usable in RPG and many shops stayed with their 10 character naming
- The reason that RPG has been limited on column name length has always been the “I” spec and it’s fixed format roots
- Since “I” specs are bypassed when using data structures for I/O, it is now possible to use the longer names
- When the ALIAS keyword is specified on the file spec, data structures using LIKEREC will have the longer alias names for subfields
- When using externally defined data structures, the ALIAS keyword is used on the data structure definition

Questions?

Thank You!

About the Presenter

Brian May is an IBM i Modernization Specialist for Profound Logic Software. He is also webmaster and coordinator for the Young i Professionals (http://www.youngiprofessionals.com). He is a husband and father of two beautiful girls. Brian can be reached at bmay@profoundlogic.com.