RPG Tricks and Techniques

Paul Tuohy
ComCon
System *i* Developer
5, Oakton Court,
Ballybrack
Co. Dublin
Ireland

Phone: +353 1 282 6230
e-Mail: paul@systemideveloper.com
Web: www.systemideveloper.com
www.ComConAdvisor.com

Paul Tuohy

Paul Tuohy, author of "Re-engineering RPG Legacy Applications" and "The Programmer's Guide to iSeries Navigator", is one of the most prominent consultants and trainer/educators for application modernization and development technologies on the IBM Midrange. He currently holds positions as CEO of ComCon, a consultancy firm based in Dublin, Ireland, and founding partner of System i Developer, the consortium of top educators who produce the acclaimed RPG & DB2 Summit conference. Previously, he worked as IT Manager for Kodak Ireland Ltd. and Technical Director of Precision Software Ltd.

In addition to hosting and speaking at the RPG & DB2 Summit, Paul is an award-winning speaker at COMMON, COMMON Europe Congress and other conferences throughout the world. His articles frequently appear in iProDeveloper, The Four Hundred Guru, RPG Developer and other leading publications. Paul also hosts the popular *iTalk with Tuohy* podcast interviews.

Agenda

System i Developer
The Summit of Education Excellence
ComCon

A smörgåsbord of some of my (currently) favourite Tips and Techniques

The Multiple Faces of RPG

```
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+..
C      HRS             IFLE       40
C      HRS             MULT(H)    RATE           PAY                7 2
C                      ELSE
C      RATE            MULT       40             REGPAY             7 2
C      HRS             SUB        40             HRSOT              3 0
C      RATE            MULT       1.5            OTRATE             3 2
C      HRSOT           MULT(H)    OTRATE         OTPAY              7 2
C      REGPAY          ADD        OTPAY          PAY                7 2
C                      ENDIF
```

```
CL0N01Factor1++++++Opcode(E)Extended-factor2++++++++++++++++++++++
C                   if        (hours <= 40)
C                   eval(H)   weeklyPay = rate * hours
C                   else
C                   eval(H)   weeklyPay = ((hours-40) * (rate*1.5))
C                                       + (rate * 40)
C                   endIf
```

```
      if (hours <= 40);
          eval(H) weeklyPay = rate * hours;
      else;
          eval(H) weeklyPay = ((hours - 40) * (rate * 1.5)) + (rate * 40);
      endIf;
```

Basic Guidelines

Be Free

- Code EVERYTHING in free form
- Even if you are adding two lines of code to an old RPG II style program

Mind your language

- Use proper names
  - customerName not custNam not cusNo not wkCsNo
  - messageType not msgType
- Speak the same language as the rest of the world
  - Table instead of Physical File
  - Index instead of Logical File

Learn and adopt from other languages

- /INCLUDE instead of /COPY
- Constants in capitals

Use the modern tool set (RDi or miWorkplace or ILEditor)

- The tools will help with everything new

Embrace what is new

- Learn new habits

Learn a new language

- It will help your RPG

Code the norm...

Program the exception

```
if (rate <> 0);
   value = value/rate;
else;
   value = 0;
endIf;
```

Old RPG habits die hard

```
monitor;
   value = value/rate;
on-error;
   value = 0;
endMon;
```

CTL-OPT (Was H Spec)

The CTL-OPT plays a key role for programs

- Defaults for program
- Debugging
- Compiler options

Use a copy member for any standard H-specs, not a Data Area

On every CTL-OPT…

```
ctl-Opt option(*srcStmt: *noDebugIO) extBinInt(*yes) CCSID(*char:*jobRun);
```

option(*srcStmt :*noDebugIO)

- Match program statement numbers with source line numbers !!
  - Easier for debugging and end user support
- Skip the individual field steps for Input and Output specs
  - Faster step function during debugging

extBinInt(*yes)

- Externally defined Binary (DDS), SMALLINT, BIGINT, INTEGER (SQL) defined as integer as opposed to default of binary

CCSID(*char:*jobRun)

- Ensure correct casting to UCS2
  - Default is *MIXED

```
ctl-Opt option(*srcStmt: *noDebugIO) extBinInt(*yes) CCSID(*char:*jobRun)
        decPrec(63) copyRight('This is mine, all mine!');
```

Compiler Options on CTL-OPT

Many CRTBNDRPG/CRTRPGMOD keywords can be specified on CTL-OPT

- The compile options specified will override the ones specified on the CRTxxxxxx command

Unsupported keywords:

- DBGVIEW, OUTPUT, REPLACE, DEFINE, PGM, SRCFILE, SRCMBR, TGTRLS


```
ctl-Opt option(*srcStmt: *noDebugIO) extBinInt(*yes) CCSID(*char:*jobRun)
        dftActGrp(*no) actGrp('COOL_GROUP') bndDir('MYBNDDIR');
```

extProc Keyword

extProc(name)

- The call will be a bound procedure call that uses the external name specified by the keyword

```
dcl-pr get_stuff likeDs(lotsOfStuff)
                 extproc('get_theInfusionEngineAndTheFluxCapacitor');
end-pr;
```

extProc(*dclCase)

- The case of the external name is the exact same as the subprocedure name

```
dcl-pr get_otherStuff extProc('get_otherStuff') end-pr;
// is the same as
dcl-pr get_otherStuff extProc(*dclCase) end-pr;
```

Specify extProc on the Procedure Interface for internal subprocedures

- Those that don't have a prototype

```
dcl-Proc calculateTax export;
  dcl-pi *n extProc(*dclCase) end-pr;
  calculateDeductions();
end-Proc;
```

Defining Subprocedures Without extProc(*dclCase)

Names can be “difficult” to read

- When looking at the call stack

```
PAYCALC1    TESTSTUFF                    _QRNP_PEP_PAYCALC1
PAYCALC1    TESTSTUFF      600           PAYCALC1
PAYCALC     TESTSTUFF      800           CALCULATEPAY
PAYCALC     TESTSTUFF      1400          GETINCOME
PAYCALC     TESTSTUFF      2000          CALCULATETAX
PAYCALC     TESTSTUFF      2800          CALCULATEDEDUCTIONS
```

- When looking at a service program exports

```
                    Procedure Exports:

Procedure Name                                              ARGOPT
CALCULATEDEDUCTIONS                                         *NO
CALCULATEPAY                                                *NO
CALCULATETAX                                                *NO
GETINCOME                                                   *NO
```

Defining Subprocedures With extProc(*dclCase)

Names “easier” to read

- When looking at the call stack

```
PAYCALC1    TESTSTUFF                    _QRNP_PEP_PAYCALC1
PAYCALC1    TESTSTUFF      600           PAYCALC1
PAYCALC     TESTSTUFF      800           calculatePay
PAYCALC     TESTSTUFF      1400          getIncome
PAYCALC     TESTSTUFF      2000          calculateTax
PAYCALC     TESTSTUFF      2800          calculateDeductions
```

- When looking at a service program exports

```
                    Procedure Exports:

Procedure Name                                                ARGOPT
calculateDeductions                                           *NO
calculatePay                                                  *NO
calculateTax                                                  *NO
getIncome                                                     *NO
```

Static

The oft ignored STATIC keyword

- Used in subprocedure declarations

A subprocedures memory is reallocated on every call

- Except for data defined with the STATIC keyword

STATIC mean that fields...

- That are specific to the subprocedure and whose state needs to be maintained between calls, do NOT have to be defined globally


```
dcl-Proc putSource2.
   dcl-Pi *n end-pi;

   dcl-s nextSeq  int(10) static;

   nextSeq += 1;
   s2.srcSeq = nextSeq;
   write source2 s2;
   return;

end-Proc;
```

Varying Length Fields

Excellent for string handling

- Reduces requirement for %TRIMx functions
- The length is set when data is moved into it

The %LEN function can also be used to reset the length

Be careful using *blanks


```
dcl-s firstName    char(10)    inz('Paul');
dcl-s lastName     char(10)    inz('Tuohy');

dcl-s v_firstName  varChar(10) inz('Paul');
dcl-s v_lastName   varChar(10) inz('Tuohy');

dcl-s fullName     varChar(21);

fullName = %trimR(firstName) + ' ' + %trimR(lastName);
fullName = v_firstName + ' ' + v_lastName;

fullName = '';
%len(fullName) = 0;
fullName = *blanks;            // fullName now has 21 spaces
```

%size and %len with Varying Length Fields

Varying length fields have two components

- The current length of the field
  - a two byte binary (5i 0) if <= 65,535
  - a four byte binary (10i 0) if > 65,535
- The actual data

%size includes the space required for the length integer

%len indicates the length of the content


```
DSPLY  Size of bigVar is: 100004
DSPLY  Size of smallVar is: 102
DSPLY  Len of bigVar is: 3
DSPLY  Len of smallVar is: 3
```

```
dcl-s bigVar   varChar(100000) inz('abc');
dcl-s smallVar varChar(100)    inz('abc');

dsply ('Size of bigVar is: ' + %char(%size(bigVar)));
dsply ('Size of smallVar is: ' + %char(%size(smallVar)));
dsply ('Len of bigVar is: ' + %char(%len(bigVar)));
dsply ('Len of smallVar is: ' + %char(%len(smallVar)));
```

How Varying Length Fields Work

A varying length field requires two (or four) more bytes of storage then the defined length

Be careful when using varying length fields in data structures

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
00	16	T	H	I	S		I	S		T	H	E		D	A	T	A		

```
dcl-ds *n;
   webOut            varChar(2048);
     webDataLen      int(5)          overlay(webOut);
     webData         char(2048)      overlay(webOut:*next);
end-ds;

webOut = 'THIS IS THE DATA';
```

CAT Functionality with Varying Fields

System i Developer
The Summit of Education Excellence
ComCon

The fixed format CAT operation allowed you to specify trailing blanks

- F1 field is right trimmed and trailing blanks added
- Trailing blanks specified after F2
  - As a literal or constant
  - As a numeric field

Define varying field for padding

- Length should be maximum length required
- Initialize to all blanks

Set length of padding field to required trailing blanks and concatenate

```
D firstName       s                10a   inz('Paul')
D lastName        s                10a   inz('Tuohy')

C     firstName     Cat       lastName:10   fullName
```

```
dcl-s firstName  char(10)      inz('Paul');
dcl-s lastName   char(10)      inz('Tuohy');
dcl-s padding    varChar(500)  inz(*blanks);

%len(padding) = 10;
fullName = %trimR(firstName) + padding + %trimR(lastName);
```

Dynamic SQL Can Be Challenging

Build an SQL statement

- ```
  select workDept, empno, firstnme, lastname from employee
     where workDept  = 'A00'          and
           hireDate  >= '1963-12-05' and
           birthDate >= '1933-08-24'
     order by workDept, empno
  ```
- But any/all of the conditions in the where clause are optional

```
dcl-s getRows int(10) inz(%elem(data));
dcl-s gotRows int(10);
dcl-ds data qualified dim(500);
   workDept  char(3);
   empno     char(6);
   firstName varChar(12);
   lastName  varChar(15);
end-Ds;

dcl-s getDept  char(3);
dcl-s getHired date;
dcl-s getBirth date;

dcl-s myStatement varChar(2000);
dcl-s myWhere     varChar(100);
dcl-s pad_And     varChar(5);

dcl-c QUOTE      '''';
dcl-c ADD_AND    ' and ';
dcl-c ADD_WHERE  ' where ';
```
```

Just String The Statement

```
if (getDept <> *blanks);
   myWhere += pad_And + 'workDept = ' + QUOTE + getDept + QUOTE + ' ';
   pad_And = ADD_AND;
endIf;

if (getHired <> d'0001-01-01');
   myWhere += pad_And + 'hiredate >= ' + QUOTE +
              %char(getHired) + QUOTE + ' ';
   pad_And = ADD_AND;
endIf;

if (getBirth <> d'0001-01-01');
   myWhere += pad_And + 'birthdate >= ' + QUOTE +
              %char(getBirth) + QUOTE + ' ';
   pad_And = ADD_AND;
endIf;

if (myWhere <> '');
   myWhere = ADD_WHERE + myWhere;
endIf;
```

Just String The Statement

```
myStatement = 'select workDept, empno, firstname, lastname ' +
              ' from employee ' +
              myWhere +
              ' order by workDept, empno';

exec SQL
  prepare myDynamic from :myStatement;

if SQLCode = 0;
  exec SQL
    declare getEmployees scroll cursor for myDynamic;

  exec SQL
    open getEmployees;

  exec SQL
    fetch first from getEmployees for :getRows rows into :data;

  gotRows = SQLErrd(3);

  exec SQL
    close getEmployees;
endIf;
```

SUBSET on EXECUTE and OPEN

System i Developer
The Summit of Education Excellence
ComCon

Variable number of parameter markers can be a problem

Problem solved with SUBSET on EXECUTE and OPEN

- Requires IBM i 7.3 TR 8 / IBM i 7.4 TR 2
- Earlier solution required use of an SQL Descriptor
  - Not for the faint of heart

Check out the article

- https://www.itjungle.com/2020/06/15/guru-subset-on-execute-and-open/


```
// We want to construct the dynamic statement below - BUT
// the statement can have any permutation of the predicates for workDept and/or
//  hireDate and/or birthDate/. There will be 0 to 3 predicates in the WHERE clause

select workDept, empno, firstname, lastname from employee
   where workDept  = ?  and
         hireDate  >= ? and
         birthDate >= ?
   order by workDept, empno
```

Using SUBSET on EXECUTE and OPEN (1 of 3)


```
dcl-s getRows          int(10) inz(%elem(data));
dcl-s gotRows          int(10);
dcl-ds data qualified dim(500);
  workDept  char(3);
  empno     char(6);
  firstName varchar(12);
  lastName  varchar(15);
end-ds;

dcl-s getDept     char(3);
dcl-s getHired    date;
dcl-s getBirth    date;

dcl-s getDept_ind  int(5) inz(-7);     // "Null" indicators - Default to Not Used
dcl-s getHired_ind int(5) inz(-7);
dcl-s getBirth_ind int(5) inz(-7);

dcl-s myStatement varchar(2000);
dcl-s myWhere     varchar(100);
dcl-s pad_And     varchar(5);

dcl-C ADD_AND    ' and ';
dcl-C ADD_WHERE  ' where ';
dcl-C ADD_DEPT   ' workDept = ? ';
dcl-C ADD_HIRE   ' hiredate >= ? ';
dcl-C ADD_BIRTH  ' birthdate >= ? ';
```

Using SUBSET on EXECUTE and OPEN (2 of 3)


```
if (getDept <> *blanks);
  myWhere = ADD_DEPT;
  pad_And = ADD_AND;
  getDept_ind = 0;        // 0 means use the value
endIf;

if (getHired <> d'0001-01-01');
  myWhere += pad_And + ADD_HIRE;
  pad_And = ADD_AND;
  getHired_ind = 0;
endIf;

if (getBirth <> d'0001-01-01');
  myWhere += pad_And + ADD_BIRTH;
  pad_And = ADD_AND;
  getBirth_ind = 0;
endIf;

if (myWhere <> '');
  myWhere = ADD_WHERE + myWhere;
endIf;
```

Using SUBSET on EXECUTE and OPEN (3 of 3)


```
myStatement = 'select workDept, empno, firstname, lastname ' +
              ' from employee ' +
              myWhere +
              ' order by workDept, empno';

exec SQL
  prepare myDynamic from :myStatement;

if SQLCode = 0;
  exec SQL
    declare getEmployees scroll cursor for myDynamic;

  exec SQL
    open getEmployees using SUBSET :getDept  :getDept_ind,
                                   :getHired :getHired_ind,
                                   :getBirth :getBirth_ind;

  exec SQL
    fetch first from getEmployees for :getRows rows into :data;

  gotRows = SQLErrd(3);

  exec SQL
    close getEmployees;
endIf;
```

Naming SQL Cursors

Use the same name as the containing subprocedure

```
dcl-proc myCoolProcedure export;
  dcl-pi *n;
    data likeDs(myData) dim(2000);
  end-pi;

  dcl-s  getColumns  uns(10) inz(%elem(data));

  exec SQL
    declare cursor myCoolProcedure scroll cursor for
      select column1, column2, column3
        from some_table
      order by column1
      for read only;

  exec SQL
    open myCoolProcedure;

  exec SQL
    fetch first from myCoolProcedure
      for :getColumns rows into :data;

  exec SQL
    close myCoolProcedure;

end-proc;
```

The Need for Overloading

```
dcl-pr format_from_Date varChar(10) extProc(*dclCase);
  dateIn date(*ISO) const;
end-pr;

dcl-pr format_from_Number varChar(10) extProc(*dclCase);
  dateIn zoned(8) const;
end-pr;

dcl-pr format_from_Character varChar(10) extProc(*dclCase);
  dateIn char(10) const;
end-pr;
```

```
dcl-s forDate date(*ISO) inz(D'2020-08-01');
dcl-s forNum  zoned(8)   inz(20200801);
dcl-s forChar char(10)   inz('2020-08-01');

dcl-s returnDate char(10);

returnDate = format_from_Date(forDate);
returnDate = format_from_Number(forNum);
returnDate = format_from_Character(forChar);
```

An Overloaded Prototype


```
dcl-pr format_from_Date varChar(10) extProc(*dclCase);
  dateIn date(*ISO) const;
end-pr;

dcl-pr format_from_Number varChar(10) extProc(*dclCase);
  dateIn zoned(8) const;
end-pr;

dcl-pr format_from_Character varChar(10) extProc(*dclCase);
  dateIn char(10) const;
end-pr;

dcl-pr format_Date varChar(10) overLoad( format_from_Date
                                       : format_from_Number
                                       : format_from_Character);
```

```
dcl-s forDate date(*ISO) inz(D'2020-08-01');
dcl-s forNum  zoned(8)   inz(20200801);
dcl-s forChar char(10)   inz('2020-08-01');

dcl-s returnDate char(10);

returnDate = format_Date(forDate);
returnDate = format_Date(forNum);
returnDate = format_Date(forChar);
```

Totally Different Parameters


```
dcl-pr format_from_Date_With_Format varChar(10) extProc(*dclCase);
  dateIn   date(*ISO) const;
  toFormat char(4)    const;
end-pr;

dcl-pr format_Date varChar(10) overLoad( format_from_Date
                                       : format_from_Number
                                       : format_from_Character
                                       : format_from_Date_With_Format);
```

```
dcl-s forDate date(*ISO) inz(D'2020-08-01');
dcl-s forNum  zoned(8)   inz(20200801);
dcl-s forChar char(10)   inz('2020-08-01');

dcl-s returnDate char(10);

returnDate = format_Date(forDate);
returnDate = format_Date(forNum);
returnDate = format_Date(forChar);
returnDate = format_Date(forDate: '*MDY');
```

Overloading Rules

These are the rules that apply when defining a prototype with the OVERLOAD keyword:

- Terminology
  - The name of the prototype containing the OVERLOAD keyword is the overloaded prototype
  - The subprocedure names listed in the OVERLOAD keyword are the candidate prototypes
- You cannot specify parameters for a prototype with the OVERLOAD keyword
- A prototype with the OVERLOAD keyword does not end with END-PR
- All the candidate prototypes must have the same (or no) return value type as the overloaded prototype
- The only other keywords allowed for the overloaded prototype are related to the data type of the return value
- The candidate prototypes can be any type of prototype
  - They can be for programs, procedures, and Java methods

Check out the article

- https://www.itjungle.com/2020/10/05/guru-overloading-subprocedures/

SQL Never "Fails"

All errors are trapped

- As if an (E) extender was being used

Can use a procedure to have "unexpected" errors make the program fail

- Procedure doubles in checking EOF


```
monitor;
  exec SQL
    insert into empa values(1, 'Paul');
  check_SQLState();

  exec SQL
    insert into empa values(1, 'Fred');
  check_SQLState();

  exec sql
    fetch next from C001 into :data ;

  if (check_SQLState());
     // It was EOF
  endIf;
on-error;
  dsply 'Caught the SQL error';
endMon;
```

SQL Exception Handler (1 of 2)

Use GET Diagnostics to determine the “success” of the last SQL statement

Set conditions for a message being sent

```
 // Get last state
exec SQL
  get diagnostics condition 1 :lastState = RETURNED_SQLSTATE;

 // All OK - just return
if (status_SQL = W_SUCCESS);

 // EOF - return true - but no message
elseIf (status = W_EOF);
   status = *on;

 // Warning - send Diagnostic message
elseIf (status_SQL = W_WARNING);
   messageType = W_DIAGNOSTIC;
   exec SQL
     get diagnostics condition 1 :messageText = MESSAGE_TEXT;

 // Anything else - send an Escape message
else;
   messageType = W_ESCAPE;
   exec SQL
     get diagnostics condition 1 :messageText = MESSAGE_TEXT;
   status = *on;

endIf;
```

SQL Exception Handler (2 of 2)

Sending an Escape message call the procedure to fail

- Which will cause an error in the calling program

Full code in notes

- Article Embedded SQL Exception/Error Handling
  http://www.itjungle.com/fhg/fhg040214-story01.html


```
if (messageType <> *blanks);
   messageText = lastState + ' ' + messageText;
   sendProgramMessage( W_MSGID
                     : W_MSGF
                     : messageText
                     : %len(%trimr(messageText))
                     : messageType
                     : W_STACK_ENTRY
                     : W_STACK_COUNT1
                     : messageKey
                     : APIError );
endif;

return status;
```

Prototype

```
        //**/ @desc Check SQL status code. <br />
        //          Checks the status code of the previoiusly executed SQL
        //          statement and, depending on the status, will send a message
        //          to the caller. <br />
        //          if the status is a warning (SQLSTATE='01nnn'/SQLCODE>0), a
        //          diagnostic message is sent to the caller <br />
        //          if the status is an error(SQLSTATE='03nnn'/SQLCODE<0), an
        //          escape message is sent to the caller <br />
        //          The function also indicates if "Row Not Found" is detected

        //    @author Paul Tuohy
        //    @return Status   <br />
        //              True if BAD error (but escape message has been sent, so
        //              caller should fail) <br >
        //              True if "Row not Found" - SQLSTATE='02nnn'/SQLCODE=100

        //    @category utility
        //    @category SQL
        //    @category error

D check_SQLState  PR             n   extProc('check_SQLState')
```

```
H noMain option(*srcStmt: *noDebugIO)
   // ***NOTE*** Module must be in the same AG as the caller

 /include prototypes

P check_SQLState  B                   export
D                 PI              n

  // Standard API Error Data Structure
d APIError        DS                  qualified
d  bytesProvided                10i 0 inz(%size(APIError))
d  bytesAvail                   10i 0 inz(0)
d  msgId                         7a
d                                1a
d  msgData                     240a

  // Prototype for QMHSNDPM (Send Program Message) API
D sendProgramMessage...
D                 PR                  extPgm('QMHSNDPM')
D   messageID                    7a   const
D   messageFile                 20a   const
D   messageData                256a   const
D   messageDataLength...
D                               10i 0 const
D   messageType                 10a   const
D   callStackEntry...
D                               10a   const
D   callStackCount...
D                               10i 0 const
D   messageKey                   4a
D   errorCode                        likeds(APIError)
```

```
  // Work fields
D messageKey      s              4a
D messageType     s             10a
D messageText     s           1024a
D status          s              n

D                 DS
D  lastState                     5a
D  status_SQL                    2a   overLay(lastState)

  // Constants
D W_DIAGNOSTIC    C                   '*DIAG'
D W_EOF           C                   '02'
D W_ESCAPE        C                   '*ESCAPE'
D W_MSGF          C                   'QCPFMSG   *LIBL'
D W_MSGID         C                   'CPF9897'
D W_STACK_ENTRY   C                   '*'
D W_STACK_COUNT1  C                   1
D W_SUCCESS       C                   '00'
D W_WARNING       C                   '01'
 /free
   // Get last state
  exec SQL
    get diagnostics condition 1 :lastState = RETURNED_SQLSTATE;
```

```
 // All OK - just return
if (status_SQL = W_SUCCESS);

// EOF - return true - but no message
elseIf (status = W_EOF);
   status = *on;

 // Warning - send Diagnostic message
elseIf (status_SQL = W_WARNING);
   messageType = W_DIAGNOSTIC;
   exec SQL
     get diagnostics condition 1 :messageText = MESSAGE_TEXT;

 // Anything else - send an Escape message
else;
   messageType = W_ESCAPE;
   exec SQL
     get diagnostics condition 1 :messageText = MESSAGE_TEXT;
   status = *on;

endIf;
```

```
     if (messageType <> *blanks);
         messageText = lastState + ' ' + messageText;
         sendProgramMessage( W_MSGID
                           : W_MSGF
                           : messageText
                           : %len(%trimr(messageText))
                           : messageType
                           : W_STACK_ENTRY
                           : W_STACK_COUNT1
                           : messageKey
                           : APIError );
      endif;

      return status;
    /end-free
    P                 E
```

Based Data Structures

System i Developer
The Summit of Education Excellence
ComCon

A Based DS can be defined using TEMPLATE keyword

- TEMPLATE allows subfields to be initialized
- Cannont “accidently” reference subfields in template


```
dcl-Ds baseAddress template qualified;
   street1  char(30);
   street2  char(30;
   city     varChar(20);
   state    char(2)     inz('TX');
   zip      char(5);
   zipPlus  char(4);
end-ds;

dcl-ds invoiceInfo qualified;
   mailAddr likeDS(baseAddress) inz(*likeDS);
   shipAddr likeDS(baseAddress) inz(*likeDS);
end-ds;
```

Referencing Based Structures

Where to store based structured

- In same member as prototypes
  - If structure relates to a program or procedure
- In “standard” copy members

Should be included in ALL programs/procedures

- Same as prototypes


```
/include QINCLUDES,BASEINFO

dcl-ds invoiceInfo qualified;
  mailAddr likeDS(baseAddress) inz(*likeDS);
  shipAddr likeDS(baseAddress) inz(*likeDS);
end-ds;
```

Overlaying a DS & Unnamed Fields

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

- Note that the DS subfields do not need to be named
- But can still have INZ values!


```
dcl-ds compileData;
   *n  char(9) inz('January')
   *n  char(9) inz('February')
   *n  char(9) inz('March')
   *n  char(9) inz('April')
   *n  char(9) inz('May')
   *n  char(9) inz('June')
   *n  char(9  inz('July')
   *n  char(9  inz('August')
   *n  char(9  inz('September')
   *n  char(9) inz('October');
   *n  char(9) inz('November');
   *n  char(9) inz('December');

   monthNames char(9) dim(12) pos(1);
end-ds;
```

Using SORTA with Group Fields

Want to sort an array with different keys?

- Group fields can provide an answer

Be sure to specify the DIM at the group field level

- Then the array can be sorted on any of the subfields

Make use of the %SUBARR() BIF

- Subset of the elements of the array


```
dcl-Ds addressInfo;
  // Note that Dim is specified at the group field level
  addressData  char(52)   dim(1000);
    streetA     char(30)   overlay(addressData);
    cityA       char(20)   overlay(addressData: *next);
    stateA      char(2)    overlay(addressData: *next);
end-Ds;

sortA %subArr(cityA:1:noLoaded);     // Sort into City sequence

sortA %subArr(stateA:1:noLoaded);    // Sort in State sequence
```

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.

The %SUBARR BIF was introduced in V5R3. In this example NoLoaded contains the number of elements loaded to the array.

An Approach to Handling Error Messages

Traditional Approach

- ERRMSGID in DDS
- Use a Message Subfile
  - Send messages to program message queue
  - Message subfile displays messages in program message queue

Unfortunately, this approach will not work in non green screen environments

- Not every client will know what a program message queue is!

An alternative approach

A module that stores messages

- Stores an array of messages
- Contains subprocedures to
  - Add a message
  - Clear stored messages
  - Retrieve a message
  - Return the number of stored messages

The full details in "Getting the Message - Parts 1 and 2"

- http://www.itjungle.com/fhg/fhg101409-story01.html
- http://www.itjungle.com/fhg/fhg102109-story01.html

Using Message Procedures

A quick idea of how it works

```
dcl-Ds def_MsgFormat qualified template;
   msgId      char(7);
   msgText    char(80);
   severity   int(10);
   help       char(500);
   forField   char(10);
end-Ds;
```

```
dcl-Ds message likeDS(def_MsgFormat) end-Ds;

clearMessages();
addMessage('ERR0001': employeeID: %char(salary));
addMessage(MS_END_RUN);
addMessage('ERR0002': employeeID);
addMessage('ERR9001': *omit: % employeeID);
addMessageText('Bad things happened!!!');
if (messageCount() > 0);
   for i = 1 to messageCount();
      getMessage( i: message);
      // Do what you will with the message
   endFor;
endIf;
```

Call Backs

Call Back

- Where a call is made back to a subprocedure in the calling program
- Not an alien concept
  - %HANDLER() BIF used with XML-INTO
  - qsort() and bsearch()

Scenario

- Program calls a subprocedure to calculate tax (yech!)

The Tax Calculation

- Certain amount of tax calculation is "standard"
- Additional tax calculations, depending on country
- Sub procedure written for each country tax calculation

The Tax Calculation Subprocedure

- Performs some standard calculations
- Calls back to a subprocedure in the calling program
  - Does not have to be in the calling program, as long as it is in a bound module
- Does some more calculations
- Returns

How do we identify the subprocedure to be called

- A procedure pointer

Calling Procedure Sets Required Procedure Pointer

In the calling procedure

- Set the procedure pointer for the relevant country tax calculation
- Imagine a lot more country procedures
  - Note that the interfaces are the same for each subprocedure

```
dcl-Pr calculateTaxIE  packed(15:2);
   gross    packed(15:2) const;
   taxCode  char(2)      const;
end-Pr;

dcl-Pr calculateTaxUS  packed(15:2);
   gross    packed(15:2) const;
   taxCode  char(2)      const;
end-Pr;

dcl-s tax_Ptr pointer(*proc);
```

```
if (country = 'IE');
   tax_Ptr = %paddr(calculateTaxIE);
elseIf (country = 'US');
   tax_Ptr = %paddr(calculateTaxUS);
endIf;

calculateTax(data: tax_Ptr);
```

The Tax Calculation Subprocedure

Generic prototype to call back to country tax subprocedure

- EXTPROC identifies a procedure pointer
- Prototype definition of all subprocedures must be the same
- Set the procedure pointer
- Call the subprocedure

```
dcl-s p_calculateTax  pointer(*proc);
dcl-Pr calculateCountryTax packed(15:2) extProc(p_calculateTax);
   gross    packed(15:2) const;
   taxCode  char(2)      const;
end-Pr;
```

```
dcl-Pi calculateTax;
   data       likeDs(b_tax_data)
   tax_Ptr    poiner(*proc) const ;
end-pi;
```

```
  // Lots of nasty tax calculations
  // Perform call back
  p_calculateTax = tax_Ptr;
  calculateCountryTax(b_tax_data.gross: b_tax_data.taxCode);
  // More nasty tax calculations
```

Summary

There you have it!

I hope you find a few of the tips and techniques useful and applicable

iTalk with Tuohy

Check it out at https://techchannel.com/italk-podcast