10 Nested Data Structures

The LIKEDS and LIKEREC keywords provide the ability to create Qualified Data Structures based on the format of existing data structures or record formats:

```
D CL_Date         DS
D  Century                       1S 0
D  Year                          4S 0
D  Month                         2S 0
D  Day                           2S 0

D myDate          DS                     LikeDS(CL_DATE)
```

This is extremely valuable to RPG IV development, but it doesn't stop there. RPG IV also supports data structures within data structures. These are known as Nested Data Structures and, strictly speaking, they are simply qualified data structures within qualified data structures —that is, a subfield of a qualified data structure may also be another qualified data structure:

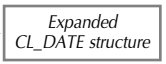
Nesting Data Structures within One Another

```
D CustOrder       DS                     Qualified Inz
D  CustNo                        7P 0
D  OrdNbr                        7P 0
D  OrdDate                             LikeDS(CL_Date)
D  BalDue                       11P 2
```

In the example above, the CUSTORDER data structure is a typical data structure, and it is Qualified. The first two subfields, CUSTNO and ORDNBR, and the last subfield, BALDUE, are traditional subfields. However, the third subfield, ORDDATE, is declared using the LIKEDS keyword, and this creates a nested data structure.

Nested Data Structure allows a subfield to be declared as a data structure—that is, the subfields from the CL_DATE data structure are used as subfields of the CUSTORDER.ORDDATE subfield. When compiled, the CUSTORDER data structure is expanded as follows:

```
D CustOrder       DS                    Qualified Inz
D  CustNo                          7P 0
D  OrdNbr                          7P 0
D  OrdDate                            Qualified
D   Century                        1S 0
D   Year                           4S 0
D   Month                          2S 0
D   Day                            2S 0
D  BalDue                         11P 2
```



The subfields from CL_DATE have been inserted after ORDDATE. The Qualified keyword, which is implied by a LIKEDS keyword, is also inserted. These new subfields are accessible using qualified syntax:

```
nYear = CustOrder.OrdDate.Year;
```

The YEAR subfield is qualified to its parent data structure, ORDDATE. ORDDATE, which itself is a subfield, is further qualified to *its* parent data structure, CUSTORDER.

There is no practical limit to the number of levels for nested data structures. Although the RPG limit of 99 nesting levels on just about any "nested" component (parens, IF statements, expressions, etc.) seems to apply to nested data structures as well.

Now the crazy part: *You can actually declare a nested data structure as an array.*

```
D Contact         DS                    Qualified
D  Name                           30A
D  phone                          10S 0 Dim(3)
D  email                          64A   Dim(3)

D Customer        DS                    Qualified Inz
D  CustNo                          7P 0
D  Contact                            LikeDS(CL_Date) Dim(6)
D  Addr                           30A
D  Addr2                          30A
D  City                           20A
D  State                           2A
```

In this example, there are 6 elements of the CONTACT subfield in the CUSTOMER data structure. This provides the ability to hold up to 6 contacts—name, phone numbers, and email addresses. The cool thing is that the CONTACT data structure's PHONE and EMAIL subfields are also arrays of 3 elements each. This means we can have up to 6 contacts per customer with up to 3 phone numbers or email addresses for each contact.

So how do you get to the second phone number of the 3rd contact? By using Qualified syntax, as follows:

```
bestNbrToCall = customer.contact(3).phone(2);
```

This retrieves the second phone number of the third contact.

Nested data structures allow you to move beyond using the OVERLAY keyword in data structure definitions by providing the ability to declare a data structure subfield as a data structure itself.

Nested-data-structure syntax *may* be used in free format and in the Extended Factor 2 syntax. It may *not* be used with traditional opcodes in Factor 1, Factor 2, or in the Result field. It is also not supported in input or output specifications.

11 *ALL and *ALLX 'xx'—The Repeating Constants

Figurative Constants such as **BLANKS* and **ZEROS* have been widely used. But most programmers overlook **ALL*, which is a very useful figurative constant. *ALL is provided in two versions:

- One replicates a literal—specified in *character* notation—matching the corresponding variable's size.
- The other replicates a literal—specified in *hexadecimal* notation—matching the corresponding variable's size. The actual value is converted to normal characters by the compiler.

The first version allows you to copy one or more characters repeatedly to the target field or to compare a field to a repeated pattern of characters. For example, to simulate *BLANKS using *ALL, you would specify *ALL' ' (*ALL followed by a blank inside single quotes).

Any series of characters may be specified following *ALL. For example, to fill a field with the letter *Q*, use *ALL'Q' on the right side of the assignment statement. If *ALL'QRST' is specified, the pattern *QRST* is repeated in the target variable. For example, if the target is 10 positions in length, then *QRST* is repeated as *QRSTQRSTQR*.

The second version allows you to build a character using hexadecimal notation. The *ALL figurative constant followed by the letter *X* (e.g., *ALLX) plus a quoted pair of hexadecimal letters or numbers can be used to build any of the 256 EBCDIC characters. But this is not limited to a single character; multiple hexadecimal characters may be specified.

*ALL and *ALLX may be used in Calculation specifications as well as the Initial Value for any field:

Initialize a Field to "Hex Zeros" with *ALLX '00'

```
D comData         S             255A   Inz(*ALLX'00')

   //  Alternatively, copy the value at runtime
C                   eval      comData = *ALLX'00'
```

To test a field for a repeating pattern of one or more characters, simply use *ALL or *ALLX in conjunction with the IF (or another conditional statement):

Test a Field for a Repeating Pattern

```
D comData         S             255A   Inz(*ALL'. ')

C                   if        comData = *ALL'. '
C                   eval      comData = *ALLX'00'
C                   endif
```

12 Embed Compiler Parameters into Source Members

One the benefits of RPG IV over RPG III and other languages is that compiler parameters from the CRTBNDRPG and CRTRPGMOD commands and a few from the CRTPGM command may be specified directly in the source member.

The *H specification*, which is often referred to as the *Header Specification*, has space in positions 7 to 80 where various controls or compiler parameters may be specified. For example, to add the activation group name *QILE* for the program, the ACTGRP keyword may be specified on the Header specification as follows:

```
     H  ACTGRP('QILE')
```

The only difference between specifying the compiler parameter on the Header specification verses the actual CRT*xxx* command is that so-called user-supplied values—such as QILE—must be enclosed in single quotes and normally must be in all uppercase (with a few exceptions). For example, ACTGRP('qile') is *not* valid, but actgrp('QILE') *is* valid.

Compiler parameter on the Header Specification are merged with those specified on the actual compiler. Header specification keywords have priority over any override compiler parameters.

Synchronize Source Statement Numbers with SEU and Debug

When RPG IV was introduced, the compiler sequenced source statements for the benefit of the compiler listing. This provided uniqueness when external definitions or /COPY members were imported.

The problem with this feature was that it became extremely difficult to match the line number from the compiler listing back to the original source member statement line number. So IBM added the OPTION(*SRCSTMT) keyword to force the compiler to use the

same line numbers on the compiler listing as in the source member. This means that line 210 in the source member is referenced as line 210 by the compiler listing.

To use OPTION(*SRCSTMT), specify it on the Header specification as follows:

Use OPTION(*SRCSTMT) to Sequence Compiler Listings

```
H OPTION(*SRCSTMT)
```

Reduce Debug Fatigue with *NODEBUGIO

If you've ever debugged an RPG IV program and were stepping through your code using the debugger's F10 (Step) function, you may have run into this problem. If you come across a READ operation to a database or display file and that file is Externally Described, the next F10 will bring you to the top of the Input specifications for the file and position you at the first field in the file. The next F10 positions you to the second field, and so on. For large or small database files, this can be annoying, particularly if you don't care about the value of the input fields.

RPG allows you to avoid this situation. Simply include the OPTION(*NODEBUGIO) keyword in the Header specification for the source member. Almost magically now, when you approach the READ operation while debugging and press the infamous F10 key, you are taken to the next executable line of code and *not* to the top of the Input specifications.

Use OPTION(*NODEBUGIO) to Avoid F10 Fatigue in Debug

```
H OPTION(*NODEBUGIO)
```

Even when no Files are declared in a source member, OPTION(*NODEBUGIO) is allowed—however, it has no effect. This is helpful for creating a standardized Header specification that may be used to /COPY into most source members.

Combine Header Specification OPTIONs

While the OPTION keyword may seem like a "singular" control, it actually supports multiple options at one time. Specify multiple options within the same OPTION keyword by separating each option with a colon. A typical OPTION keyword might include *SRCSTMT and *NODEBUGIO as follows:

Specify Multiple OPTIONs

```
H OPTION(*SRCSTMT : *NODEBUGIO)
```

The compiler keywords most often used on a Header specification include—but are not limited to—the following:

- OPTION(*NODEBUGIO : *SRCSTMT)
- BNDDIR
- DFTACTGRP(*NO)
- ACTGRP
- OPENOPT(*INZOFL)
- EXTBININT

13 Avoid "Surprise Initialize"

Legacy code is often the cause of the infamous Decimal Data Error periodically raising its ugly head. In my experience, this is usually caused by blanks or X'40' characters being stored in numeric fields, which often are actually numeric subfields of a Data Structure.

Fixing this problem can be easy if you simply initialize the data structure. To do this, add the INZ keyword to the Data Structure, and you're done. I have personally corrected around 80 percent of decimal data error problems I've encountered by simply adding the INZ keyword to the Data Structure declaration.

The INZ keyword on the Data Structure declaration specification causes its subfields to be set to zero for numeric fields, to blanks for character fields, and to the oldest (earliest) date for date/time fields. Without the INZ keyword, Data Structures—and therefore all of their subfields—are initialized to blanks.

Use INZ to Fix Decimal Data Errors

```
D MyStuff         DS                  INZ
D  ItemNo                          5P 0
D  Price                           7P 2
D  Desc                           30A
```

Blanks in data structures cause decimal data errors when the data structure contains non-character subfields.

14 Qualified Externally Described Files (1)

With the introduction of Qualified Data Structures (see Tip #8), IBM quietly added the ability to qualify input file fields and, by doing so, they sped up I/O operations.

By converting an Externally Described file to a Qualified File, the I/O operation is performed in one operation rather than a series of internal "MOVE/MOVEL" operations for each input field. To declare a Qualified File, specify the following:

- An externally described file.
- A PREFIX keyword on the File specification identifying a Qualified Data Structure.
- A Qualified, Externally Described Data Structure.

The key component here is to specify the PREFIX keyword on the File Description specification with a data structure name followed by a period; for example:

```
FCUSTMAST  IF   E           K DISK    Prefix('CM.')
```

The data structure name on the PREFIX keyword must be enclosed in quotes, be specified in all uppercase letters, and be followed by a period.

When a PREFIX keyword is used in this way, qualified field names are generated (e.g., CM.CUSTNO). To define the Data Structure named CM that is associated with the CUSTMAST File specification, a Qualified Externally Described Data Structure must be declared:

```
FCUSTMAST  IF   E           K DISK    Prefix('CM.')

D CM            E DS                  EXTNAME(CUSTMAST)
D                                     QUALIFIED
```

In the example above, the Qualified Data Structure (see Tip #8) named CM is declared. Its format (i.e., subfield) is derived from the external definition for the file named CUSTMAST. This means that all the fields from CUSTMAST are included as subfields for the CM data structure.

The PREFIX('CM.') keyword on the File specification associates the input buffer of the CUSTMAST file with the CM data structure. Hence, all input fields must be referred to using Qualified Syntax. In fact, read operations for CUSTMAST are automatically mapped into the CM data structure.

Qualified, Externally Described Files

```
FITEMMAST  IF   E           K DISK    Prefix('IM.')

D IM            E DS                  ExtName(ITEMMAST)
D                                     Qualified
 /free
       read ItemRec;
       dow NOT %EOF()
          if im.QtyOH <=0;
             joblog('Item %s out of stock.':im.item);
          endif;
          if im.backOrd > 0;
             joblog('Item %s is on backorder.':im.item);
          endif;
          if im.price <= 0;
             joblog('Warning item %s has no price.':im.item);
          endif;
          read ItemRec;
        enddo;
   *inlr = *ON;
 /end-free
```

When this technique is applied, the rule of Qualified Data Structure syntax also applies. Since the file is mapped to a qualified data structure, its input fields are renamed to IM.*xxxxxx*, where *xxxxxx* is the original field name.

The input fields must be referred to using only qualified syntax; they are no longer considered stand-alone fields.

The types of files that can use this technique include Input, Input-Add, Update, and Update-Add. Files declared as Output-only cannot use this technique.

15 Qualified Externally Described Files (2)

Qualified data structures aren't limited to data structures—you can also use them as qualified Input fields to a given input file name. Thus, you can have multiple declarations of the same file in the same program and avoid input buffer/field name conflicts. Here's how it works:

```
FCUSTMAST  IF   E           K DISK    PREFIX('CM.')
FCUSTMAST1 IF   E           K DISK    PREFIX('LGL.')
F                                     RENAME(CUSTREC : CUSTLGL)

D CM              DS                  LikeRec(CUSTREC)
D LGL             DS                  LikeRec(CUSTLGL)
D save            DS                  LikeRec(CUSTREC)
```

The Compiler uses the CUSTMAST file name to import the external definition for the CUSTMAST file. The LIKEREC keyword is used to define the format of the CM, LGL, and SAVE data structures. Data structures created by the LIKEREC keyword are Qualified Data Structures and inherit a subfield for each input field in the associated Input record format. The format name specified on LIKEREC must be from a file declared on the File specifications in this source member.

In the example above, the CM and SAVE data structures will have identical formats, whereas the LGL data structure matches the format of the CUSTMAST1's CUSTLGL record format.

The PREFIX('CM.') keyword indicates that the CUSTMAST file's fields should be mapped to a Qualified Data Structure named *CM*. Note that when this technique is applied, the prefix value must be enclosed in quotes, specified in all uppercase letters, and followed by a period.

The PREFIX('LGL.') keyword indicates that the CUSTMAST1 file's fields should be mapped to a Qualified Data Structure named 'LGL.' CUSTMAST1 is a logical file built over the CUSTMAST file and contains many, if not all, of the same field names as CUSTMAST.

When data is read from either the CUSTMAST or CUSTMAST1 files, it is automatically mapped into the CM or LGL data structures, respectively. A side-effect of this technique is that I/O performance is slightly improved. Normally, RPG copies input fields to the input buffer one field at a time. Using Qualified Externally Described files forces it to copy the entire buffer into the program in one operation.

16 Calculate the End-of-Month Date

In several types of applications, the end-of-month date is important. It's often used to determine when to run month-end reports, close the current month's books, or determine if a payment is past due.

Calculating the end-of-month date for any given date can be easily accomplished in RPG IV. One way is to use traditional fixed-format opcodes such as ADDDUR, EXTRCT, and SUBDUR (in that order). The other way uses built-in functions on one line of code. Either method is effective; both run with negligible differences in performance.

The formula for calculating the last day of the month is as follows:

- Add 1 month to the desired date X, giving a new date Y.
- Extract the day number *D*, from date *Y*.
- Subtract *D* days from date *Y* giving the end-of-month date.

These three steps can be performed in traditional RPG IV as follows:

Calculate End-of-Month in Fixed Format

```
D day             S               5I 0
D nextMonth       S                 D   DatFmt(*ISO)
D endOfMonth      S                 D   DatFmt(*ISO)
D myDate          S                 D   Inz(*SYS)

C     myDate        AddDur    1:*Months     nextMonth
C                   Extrct    nextMonth:*D  day
C     nextMonth     SubDur    day:*Days     EndOfMonth
```

In the example above, the field named MYDATE contains the date that is used to determine the end-of-month date.

On the first line, one month is added to MYDATE (a date field), giving NEXTMONTH (also a date field). If MYDATE is D'2006-11-15', then NEXTMONTH becomes D'2006-12-15'.

On the second line, the day of the month is extracted from NEXTMONTH and stored in DAY. This day is extracted from NEXTMONTH rather than from MYDATE because it could vary between those months. For example, if the original date is January 30, 2006, then adding one month would result in a new date of February 28, 2006. (If in step 3 we

extracted the day from MYDATE, 30 would be returned instead of the correct value, which is 28.)

On the third line, the day of the month is subtracted from NEXTMONTH. This returns the last day of the previous month, which is also the last day of the month of the date we original specified.

If free format is preferred, you can perform this algorithm in one statement. The use of the %MONTHS, %DAYS, and %SUBDT built-in functions allows the end-of-month routine to be performed in one expression, as follows:

Calculate End-of-Month in Free Format

```
D  myDate         S               D   Inz(*SYS)
D  EOM            S               D
 /free

    eom = (myDate+%months(1))
          %days(%subdt(myDate+%months(1):*DAYS));
 /end-free
```

The date variable *EOM* is assigned the end-of-month date. The starting date is specified in MYDATE. This expression performs the same tasks as the original fixed-format method. Either method may be used.

A GETEOM (retrieve end-of-month) subprocedure can be created using either of these two methods. This subprocedure accepts an input date and returns the end-of-month date for the input date, as follows:

Get End-Of-Month Subprocedure

```
P GetEOM          B                   Export
D GetEOM          PI              D   DATFMT(*ISO)
D  inDate                         D   Const DATFMT(*ISO)

 /free
    return (inDate+%months(1))
           %days(%subdt(inDate+%months(1):*DAYS));
 /end-free
P GetEOM          E
```

This subprocedure is a good argument for a macro language in RPG IV. In many other languages, I could have created a macro that would expand to insert the date into the expression. The expanded code would appear in-line in the program rather than in a sub-procedure.

17 Using Free-Format Comments in Fixed-Format Code

When free-format syntax was introduced to RPG IV, an alternate syntax for comments was also introduced. But unlike free-format source code, free-format comments are not required to be enclosed in /free and /end-free compiler directives.

Here's an example of free-format comments being used in free-format source code:

```
D Notes           S           4000A   Varying

 /free
     // These are some comments
        // More comments here
        if A = B;  // Inline comments too!
          bEqual = *ON
     endif;
 /end-free
```

Comments may appear anywhere in a free-format source line. However, once the free-format comments appear, no other data is recognized on that line. Therefore, they may appear on the same line *following* a free-format statement but not *preceding* them on the line.

Free-format comments may also be intermixed within fixed-format statements—that is they are not required to be stuffed between /FREE and /END-FREE directives:

Free-Format Comments in Fixed Format

```
     // Declare Lillian Date field
D nDays           S             10I 0

     // Calculate days since Oct 14 1582
C      InputDate     SubDur    BaseDate      nDays:*DAYS

     // Pass the duration to the OS/400 API.
        // Calculate the day of the week.
C                    CALLP     CEEDYWK(nDays: nDayOfWeek :*OMIT)

     // Return the day of week to the caller.
C                    return    nDayOfWeek
P GetDayOfWeek    E
```

18 Get Day-of-Week Name

Often the name of the day of the week is needed for reports, output displays, and web pages. Fortunately, OS/400 includes an API that can format the date in many ways, including returning a simple name of the day of the week.

The CEEDATE API is used to format a date as words. To convert a date into the name of the day of the week, we need to pass to the API the number of days since October 14, 1582 (the so-called "Lillian date") and a formatting string. The API returns the name of the day of the week:

GetDayName Subprocedure Source Code

```
P GetDayName      B                   Export
D GetDayName      PI            10A
D  inputDate                      D   Const DATFMT(*ISO)
D  rtnDayName                     10A   OPTIONS(*NOPASS)

D BaseDate        S                 D   INZ(D'1582-10-14')
D nDayOfWeek      S               10I 0
D nDays           S               10I 0
D szDay           S               10A

C                   TEST(E)                 inputDate
C                   if        %ERROR
C                   return    'Invalid'
C                   endif

C     inputDate     SubDur    baseDate      nDays:*DAYS

C                   CallP     CEEDATE(nDays:'Wwwwwwwwwz':szDay:*OMIT)
C                   if        %parms()>= 2
C                   eval      rtnDayname = szDay
C                   endif
C                   return    szDay
P GetDayName      E
```

In the above example, the input date is tested to verify that it is a valid date. Then it calculates the Lillian date for the input date (i.e., the number of days since October 14, 1582). That Lillian date is passed as the first parameter to the CEEDATE API.

The second parameter is a formatting code. The format 'Wwwwwwwwwz' indicates that the Day-Of-The-Week Name is to be returned. The day name is returned to the szDAY field specified on the third parameter. That day name is subsequently returned to the caller.

The CEEDATE API accepts many formatting codes, ranging from entire date as words to the abbreviated day name; for example:

Format String	Example Output
YYMMDD	880516
YYYYMMDD	19880516
YYYY-MM-DD	1988-05-16
<JJJJ> YY.MM.DD	Showa 63.05.16
<CCCC> YY.MM.DD	MinKow 77.05.16
MMDDYY	050688
MM/DD/YY	05/06/88
ZM/ZD/YY	5/6/88
MM/DD/YY	05/06/1988
MM/DD/Y	05/06/8
DD.MM.YY	09.06.88
DD-RRRR-YY	09- VI-88
DD MMM YY	09 JUN 88
DD Mmmmmmmmmm YY	09 June 88
ZD Mmmmmmmmmz YY	9 June 88
Mmmmmmmmmz ZD, YYYY	June 9, 1988
ZDMMMMMMMMZYY	9JUNE88
YYMMDDHHMISS	880516204229
YYYYMMDDHHMISS	19880516204229
YYYY-MM-DD HH:MI:SS.999	1988-05-16 20:42:29.046
WWW, ZM/ZD/YY HH:MI AP	MON, 5/16/88 08:42 PM
Wwwwwwwwwz, DD Mmm YYYY, ZH:MI AP	Monday, 16 May 1988, 8:42 PM

19 Run CL Commands from an FTP Client

Have you ever needed to run a CL command from within an FTP client when you were connected from a PC or Linux box to your iSeries or System i5? There *is* a way.

FTP clients usually support the RCMD command. This command allows you to run commands on the target/remote site from within an FTP session. For example, to run the ADDLIBLE QGPL command, the following FTP statement could be used:

```
>>  RCMD  ADDLIBLE QGPL
```

If the FTP client doesn't directly support RCMD, the FTP QUOTE command can be used. To use QUOTE to run RCMD, specify the RCMD as the parameter to QUOTE, as follows:

```
>>  QUOTE RCMD ADDLIBLE QGPL
```

The FTP QUOTE command conveys the rest of the statement that follows the QUOTE command to the remote server. If the server supports the statement, it is processed accordingly. So between iSeries systems, use RCMD, but between a PC and an iSeries, you may need to use QUOTE RCMD.

20 Put Your Program to Sleep

Inevitably, there will be times when performance isn't taken into consideration for a section of a program. For example, if a record is locked and you would like to give the user the opportunity to try to access the record again, you may want to wait 2 to 30 seconds before attempting another record access.

There are several inefficient ways to accomplish a planned hold or *wait period* in RPG IV. One common yet very poor method involves looping in a Do loop for several thousand iterations. This is arguably the worse kind of delay-loop technique, as it not only delays the current user's job, it also eats up CPU cycles and system resources, consequently impacting every other active job on the system.

A better approach is to use the system's interrupt capabilities and put the job to sleep for the desired wait period. There are two efficient techniques to accomplish this: the sleep() C runtime function or the waittime() MI built-in. My preference is the C runtime function, as it is the easier to use.

Both these functions avoid eating up valuable CPU resources. In fact, there should be virtually no CPU utilization as a result of using either the sleep() or waittime() functions.

As with any C or MI function, a prototype must be created before you can use it. The sleep() function from the C language prototype follows:

RPG IV Prototype for the sleep() C Runtime Function

```
D sleep           PR            10U 0 extProc('sleep')
D  milliSecs                    10U 0 value
```

Once prototyped, call sleep(*xxxx*)—where "*xxxx*" is the number of seconds for which you want your program to sleep—from within RPG IV. To sleep for 2 seconds, specify a value of 2000, as follows:

```
callp  sleep(2000);
```

21 Use VARYING to Improve Performance

Fixed-length character fields have been around forever. With RPG IV, new varying-length fields were introduced. These new fields allow you to store character data in the same way as fixed-length fields *except* the fields keep track of their "current" length.

By tracking their own length, VARYING fields can help out with optimization by informing RPG IV's opcodes of its current length. That way, they perform their task only on the current length instead of the entire field. This is particularly useful with the so-called "string" built-in functions—%SCAN, %XLATE, %CHECK, and so on—as well as simple assignment statements (e.g., EVAL operations).

For example, assume you have a 4,000-byte character field. Scanning, translating, comparing, clearing that field always occurs on all 4,000 bytes. But when a VARYING field that is defined as 4,000-bytes in length is used, only its current length is scanned, translated, compared, or cleared. If its current length is 15, only 15 bytes are manipulated instead of the entire 4,000.

This in and of itself isn't a huge performance benefit; however, when it is compounded several thousand times in a loop or other routine, it matters very much. So when processing many database records or processing subfile or web browser (HTML) output in a loop, adding the VARYING keyword to a character field can improve performance.

VARYING fields are also compatible with the DB2/400 VARLEN keyword. Database files with fields that have the VARLEN keyword automatically map to VARYING fields in RPG IV.

A couple of points about VARYING fields:

- The entire length of the field is allocated by the compiler at startup.
- A 5U0 (2-byte integer) hidden prefix is stored with the field. It contains the current length of the field.
- When the current length of the field is shortened, the current length is changed but the data is not changed.
- When the current length of the field is increased, the current length is changed, and the bytes following the previous current-length position are cleared to the new current length.

To illustrate VARYING fields, the following 10-position VARYING field named VTEXT occupies 12 bytes of memory. This comprises its 2-byte integer prefix and the 10 bytes of the field itself. If it were initialized to the value 'Cozzi', it would be declared as follows:

```
D VText           S             10A   Varying Inz('Cozzi')
```

This field would have the following memory structure:

		Memory/Byte Positions									
1	2	3	4	5	6	7	8	9	10	11	12
Data Positions →		1	2	3	4	5	6	7	8	9	10
Current Length		**Data**									
		C	**o**	**z**	**z**	**i**					
0	0	C	9	A	A	8	4	4	4	4	4
0	5	3	6	9	9	9	0	0	0	0	0

The following illustrates declaring and using a 4,000-position VARYING field with the %SCAN built-in function.

VARYING Keyword to Help Improve %SCAN Performance

```
D Notes           S           4000A   Varying

 /free
     notes = 'Customer was very happy with the +
              product and will tell their friends.';
     if (%scan('not happy' : notes) > 0);
       bHappyCust = *ON;
     endif;
 /end-free
```

In the example above, the %SCAN built-in function runs substantially faster thanks to the NOTES field being defined as a VARYING field. Only the current length, about 70 bytes, is scanned. If the entire 4,000 bytes were to be scanned, it would (obviously) take more time.

Most opcodes and built-in functions take advantage of VARYING fields. Every opcode that supports traditional fixed-length fields also supports VARYING-length fields.

22 Converting Numeric to Character with %CHAR

In /free RPG IV syntax and with the EVAL operation, the question of how to convert numeric to character keeps coming up. In traditional fixed-format RPG, the MOVE opcode was used, plain and simple—but what about /free?

RPG IV includes the %CHAR built-in function. This built-in function converts just about any non-character data-type to character, and it does it extremely easily.

The syntax is *myData* = %char(*myNumVal*). This converts the numeric value in the myNumVal field to character(s) and stores the value, left-justified and zero-suppressed, in the myNumVal field.

%CHAR also strips off leading zeros and left-justifies the result. It will include a negative sign if applicable but not a positive sign. In addition, the right side of the decimal notation always retains its zeros. So if the value being converted is declared as a 9P4 value, the right side of the decimal always includes 4 digits.

Convert Numeric to Character

```
D price           S              7P 2 Inz(12.50)
D szText          S             10A

   // Convert to Character: szText = '12.50'
C                   eval      szText = %char(price)

   // Or in /free syntax, as follows:
 /free
    szText = %char(price);
 /end-free
```