

Session:

# CL at 7.1 and 7.2 (CL Compiler Enhancements from V5R3 to now)

Guy Vig

Guy.Vig@gmail.com

*i want stress-free IT.  
i want control.  
i want an i.*

Copyright Guy Vig 2014

## Overview of this session

With the exception of providing an ILE CL compiler in V3R1, very little about the operating system control language (CL) **compiler** changed from release 1 in 1988 through release V5R2 in 2002.

In this session, I will cover all of the **CL compiler** enhancements made starting in V5R3 up through the very latest enhancements delivered as PTFs to release 7.1 of IBM i as well as the 7.2 release which became generally available in May of 2014.

There were other CL-related enhancements made to the operating system but were **not** specific to the CL compiler. Those are covered in a different session; there's plenty of **CL compiler** enhancements to fill this session.

Attendees for this session should get an understanding of what CL compiler enhancements are available and some of the background on why those enhancements were chosen.

# “General” CL Commands

- New/changed IBM CL commands in **every** release
  - For **V5R3**: **57** new, **247** changed commands
  - For **V5R4**: **43** new, **151** changed commands
  - For **6.1**: **141** new, **319** changed commands
  - For **7.1**: **38** new, **179** changed commands
  - For **7.2**: **70** new, **165** changed commands

## Notes:

There has been ongoing investment in the set of CL commands shipped with the operating system and other IBM licensed programs that run on the IBM i operating system. With the increase in the number of callable application programming interfaces (APIs), this has diminished somewhat. However, we've never shipped a release that didn't have new and changed CL commands. We've also cut back on the number of new "Work with"-type commands to encourage interactive system administration through our graphical system tools, such as Navigator for i.

In V5R3, we continued to invest in both new CL commands and updated and enhanced the rich set of existing CL commands. There were over 50 new commands and over four times that many CL commands which were updated for V5R3.

Likewise in V5R4, 6.1, 7.1, and 7.2 releases, there are many new commands and several times that many changed commands.

What's different is that, starting with V5R3, several of those new and changed commands were ones that are only valid in CL programs. After many, many years and releases of no enhancements, V5R3 marked the first release of major improvement to CL as an application development language.

# Binary (Integer) Variables

- New TYPE values of \*INT and \*UINT added to the DCL command
- **\*\* First new CL variable data types since 1980 \*\***
- LEN(2) and LEN(4) supported
- Much "cleaner" than using %BIN built-in
- Passing parameters to APIs
- Passing parameters to other HLL programs or receiving integer parameters from other HLL programs

## Notes:

One enhancement in V5R3 was the added support for true binary integer variables. The compiler has supported the %BINARY built-in function to view character data as a two-byte or four-byte integer. This was much more awkward than simply declaring integer variables, like other HLLs. In V5R3, you can declare 2-byte or 4-byte signed or unsigned integer variables. We expect these to be used when calling APIs or other high-level language programs that expect integer parameters.

Interesting to note is that these were the first new data types since System/38 release 1 in 1980!

# Control Flow Enhancements

- **Three “flavors” of DO loop commands**
  - DOWHILE, DOUNTIL, and DOFOR
  - Up to 25 levels of DOxxx nesting supported
- **Loop control flow commands**
  - LEAVE and ITERATE
- **Case/subcase commands**
  - SELECT, WHEN, OTHERWISE, and ENDSELECT
  - Up to 25 levels of SELECT nesting supported

## Notes:

The area of control flow can be argued to be a “nice to have” for CL ... You could use combinations of IF, ELSE, and GOTO statements to simulate higher-level control flow constructs, such as DO loops. The problem was that someone reading the CL code would have to read it very carefully to understand that the code was really implementing a looping construct.

Recognizing that fact, support was added in V5R3 for three new DO loop commands:

- DOWHILE
- DOUNTIL
- DOFOR

In addition, to facilitate cleaner control flow within loops, LEAVE and ITERATE commands were added. This avoided having to use GOTO commands within loops.

Finally, to handle a case construct in CL, support was added for four new commands: SELECT, WHEN, OTHERWISE, and ENDSELECT.

The next few slides provide a quick example of each new control flow command.

## DOWHILE Loop

- Same COND support as IF statement in CL
- Evaluates COND at "top" of loop
- Old-style coding example:

```
DCL  VAR(&LGL)  TYPE(*LGL)
      :
CHECK:  IF COND(*NOT &LGL) THEN(GOTO DONE)
      :  (group of CL commands)
GOTO CHECK
DONE:
```

- New-style coding example:

```
DOWHILE  COND(&LGL)
      : (group of CL commands) ← body will be run zero or more times
ENDDO
```

## Notes:

The new DOWHILE command lets you continue looping “while” a condition evaluates logically true.

The controlling COND parameter has the same syntax as the COND parameter on the IF command.

Like similar loop constructs in other high-level languages, the CL DOWHILE tests the controlling condition at the top of the loop. This means that the body of the DOWHILE loop will not be run at all if the controlling condition evaluates to false the first time the DOWHILE command is run.

## DOUNTIL Loop

- Similar COND support as IF statement in CL
- Evaluates COND at "bottom" of loop
- Old style coding example:

```
DCL  VAR(&LGL)  TYPE(*LGL)
      :
LOOP:
      :      (group of CL commands)
      IF COND(*NOT &LGL) THEN(GOTO LOOP)
```

- New style coding example:

```
DOUNTIL  COND(&LGL)
      : (group of CL commands) ← body will be run one or more times
      ENDDO
```

## Notes:

The new DOUNTIL command looks similar to DOWHILE.

The differences for DOUNTIL are that the loop test occurs at the bottom of the loop and looping continues until the COND parameter evaluates true. The condition test is the opposite of DOWHILE in that the loop condition is assumed to start out false.

For example, you look through records until the right one is found. Because the loop test is done at the bottom, the body of the loop is always run at least one time.

## DOFOR Loop

- Syntax:  
**DOFOR VAR( ) FROM( ) TO( ) BY( )**
- BY defaults to '1', other parameters are required
- VAR must be \*INT or \*UINT variable
- FROM and TO can be integer constants, expressions, or variables
- BY must be an integer constant (can be negative)
- FROM/TO expressions are evaluated at loop initiation; TO evaluated after increment
- Checks for loop exit at "top" of loop (like DOWHILE)

## Notes:

The new DOFOR command is a basic yet powerful counting-type loop.

The control CL variable must be an integer variable. The initial value and terminating values can be integers or integer expressions (a constant or expression which can be represented as an integer). The increment defaults to one but can be a positive or negative integer constant. We allow the increment to be zero, although that kind of defeats the purpose!

Because the terminating value can be an expression and the expression is recalculated after each increment, the terminating value can be a moving target.

Like the DOWHILE command, the DOFOR exit check occurs at the top of the loop. This means that the body of the DOFOR loop is not run at all if the FROM value is either greater than the TO value (if BY is positive or zero) or is less than the TO value (if BY is negative).

## DOFOR Loop (continued)

- Old-style coding example:

```

DCL  &LOOPCTL  TYPE(*DEC) LEN(5 0)
DCL  &LOOPLMT  TYPE(*DEC) LEN(5 0)
:
CHGVAR  &LOOPCTL  VALUE(1)
CHECK:  IF COND(&LOOPCTL *GT &LOOPLMT)  THEN(GOTO  DONE)
:      (group of CL commands)
CHGVAR  &LOOPCTL  VALUE(&LOOPCTL+1)
GOTO CHECK
DONE:

```

- New-style coding example:

```

DCL  &LOOPLMT  TYPE(*INT) LEN(4)
DCL  &LOOPCTL  TYPE(*INT) LEN(4)
:
DOFOR  VAR(&LOOPCTL)  FROM(1)  TO(&LOOPLMT)  BY(1)
:      (group of CL commands) ← body will be run zero or more times
ENDDO

```

## Notes:

This illustrates how a counting-type loop could have been done using IF and GOTO commands.

It also shows how the same loop can be coded more simply and clearly using the new DOFOR command.



## LEAVE and ITERATE

- Allowed only within a DOWHILE, DOUNTIL or DOFOR group
- LEAVE passes control to next CL statement following loop ENDDO
- ITERATE passes control to end of loop and tests loop exit condition
- Both support CMDLBL (Command label) parameter to allow jump out of multiple (nested) loops
  - Both default to \*CURRENT loop

```

LOOP1: DOFOR &COUNTER FROM(1) TO(&LIMIT)
  LOOP2: DOUNTIL COND(&FOUND)
    IF (%SST(&NAME 1 10) *EQ '*NONE') THEN(LEAVE LOOP1)
    ELSE (DO)
      IF (%SST(&NAME 11 10) *EQ '*LIBL') THEN(ITERATE)
    ENDDO
  ENDDO      /* End of DOUNTIL loop */
ENDDO      /* End of DOFOR loop */

```

## Notes:

In addition to providing DO loop CL commands, two new loop control commands were added. Users of RPG can think of these as similar to the LEAVE and ITER op codes.

A powerful feature of these new CL commands is the ability to have the command act on a loop which contains the loop where the command appears. This is done by specifying a label on the LEAVE or ITERATE command that appears on a parent DO loop command. The command LEAVE LOOP1 in this example will exit both LOOP2 and LOOP1 and control will go to the command which follows the ENDDO command for LOOP1.

If no label is specified, the LEAVE or ITERATE command is assumed to be for the DO loop where the command appears. The ITERATE command in this example will cause control to go to the ENDDO command for LOOP2 which will test the loop exit condition.

## SELECT Group

- SELECT starts a group; this command has no parameters
- ENDSELECT ends group; this command has no parameters
- Group must have at least one WHEN
  - WHEN command has COND and THEN parameters (like IF)
- OTHERWISE (optional) run if no WHEN COND = True
  - OTHERWISE command has only CMD parameter (like ELSE)
- Example:

```

SELECT
  WHEN    COND (&TYPE *EQ *CMD)  THEN (DO)
        : (group of CL commands)
  ENDDO
  WHEN    COND (&TYPE *EQ *PGM)  THEN (DO)
        : (group of CL commands)
  ENDDO
  OTHERWISE  CMD (CHGVAR  &BADTYPE  '1')
ENDSELECT

```

## Notes:

The new SELECT command provides a case structure in a more straightforward way than cascading IF THEN ELSE IF commands. If you prefer to line up your corresponding IF and ELSE commands, the SELECT structure is more readable since the WHEN commands all line up instead of walking across the page from left to right.

Within a SELECT group, the COND for the first WHEN command is evaluated. And, if it evaluates as logically true, the command associated with the THEN parameter is run and control passes to the command following the ENDSELECT. A null THEN is allowed, which causes control to go to the command following the ENDSELECT.

If the first WHEN condition is false, each successive WHEN condition is tested until one that evaluates as logically true is found. If no WHEN condition is found to be true, and there is an OTHERWISE command in the SELECT group, the command associated with the OTHERWISE is run. If there is no OTHERWISE command, control passes to the command following the ENDSELECT.

# Enhanced File Support

- Will support up to 5 file "instances" using DCLF
  - Instances can be different files or the same file
- New OPNID (Open identifier) parameter added to DCLF statement
  - Default for OPNID is \*NONE
  - Only one DCLF allowed with OPNID(\*NONE)
- OPNID accepts simple name, up to 10 characters

## Notes:

CL programmers have told us that they can use the %BINARY built-in function to work around not having binary integer variables. They can also use combinations of IF and GOTO commands to work around not having DO loops in CL. But, they also told us that they couldn't work around the CL limit of only declaring and using a single file in a CL procedure without writing multiple CL procedures.

We wanted to make a substantial improvement, but there didn't seem to be the need to work with an unlimited set of files. This allowed us to extend the Declare File (DCLF) command to support up to five files.

The OPNID specified on each DCLF command must be unique, but you can have multiple DCLF commands that refer to the same file. For example, you could have two DCLF commands for database file FILEA with OPNID values of PASS1 and PASS2. This would allow you to read the file the first time specifying OPNID(PASS1) on the RCVF command and, when end-of-file is reached, read the file a second time specifying OPNID(PASS2) on a RCVF command. Not exactly pretty, but it works!

## File Support (continued)

- If OPNID specified, declared CL variables are prefixed by this name and an underscore (e.g. &MYTESTFILE\_CUSTNAME )
- OPNID added to existing file I/O CL commands
  - RCVF
  - ENDRCV
  - SNDF
  - SNDRCVF
  - WAIT

## Notes:

To maintain upward compatibility of existing CL programs that used DCLF, we allow one instance of DCLF that does not specify an open identifier. You can have one DCLF command with OPNID(\*NONE) specified or implied plus up to four more DCLF commands with OPNID(name), or five DCLF commands with OPNID(name).

Because all of the I/O commands assumed a single file, we needed to add the open identifier to each I/O command to allow the CL programmer to identify on which file is to operate.

# Longer \*CHAR Variables

- Old limit was 9999 bytes for TYPE(\*CHAR)
- New limit is 32767 bytes for TYPE(\*CHAR)
- DCLF did not generate CL variables for character fields longer than 9999 bytes in a record format (**this support was added in V5R4**)
- Limit for TYPE(\*CHAR) and TYPE(\*PNAME) on PARM, ELEM, and QUAL command definition statements stays at 5000 bytes
- VALUE (on DCL) limited to first 5000 bytes

## Notes:

Another V5R3 enhancement allowed larger character CL variables to be declared.

The old limit was 9999 bytes. That limit increased to 32767 bytes in V5R3.

The old limit of 5000 bytes for the VALUE parameter on DCL was not changed.

Character fields in a record format referenced by a DCLF command that were longer than 9999 bytes did not generate CL variables in V5R3, but will generate TYPE(\*CHAR) CL variables in V5R4.

# More Parameters

- Previous limit was 40 for PGM and TFRCTL, and 99 for CALL command
- New limit is 255 parameters for PGM, CALL, and TFRCTL
- Limit for CALLPRC (only allowed in ILE CL procedures) stays at 300
- Number of PARM statements in a CL command increased from 75 to 99

## Notes:

Another “no workaround” limit that was raised had to do with the number of parameters that could be passed into a CL program, or passed out when the CL program called (or transferred control to) another program. The limit was 40 for many years.

The CALL command parameter limit went up to 99 just in V5R2. In V5R3, all of these parameter limits were raised to 255.

In a somewhat related change, the number of parameters that can be defined in a CL command was raised from 75 to 99.

# Passing Parmas "by value" (ILE CL only)

- CALLPRC (Call Procedure) command supports calls from ILE CL procedures to other ILE procedures
- In prior releases, CALLPRC only supported passing parameters "by reference"
- Can specify \*BYREF or \*BYVAL special value for each parameter being passed
- Enables ILE CL to call many MI and C functions and other procedure APIs
- Maximum numbers of parameters still 300

## Notes:

Up to this point, all of the CL compiler improvements added in V5R3 are true for BOTH of the CL compilers, the original CL compiler and the ILE CL compiler (added in V3R1). This last V5R3 enhancement only applies to ILE CL because the original CL compiler has no ability to call ILE procedures which are part of a service program.

If you've used any of the ILE compilers, you probably know that IBM ships thousands of procedure-level APIs. Many of those procedures are defined to expect parameters to be passed "by value" and CL could only pass parameters "by reference".

In V5R3, the CALLPRC command allows you to choose whether the parameter is passed by reference or by value.

# V5R3 Was Not a “One-Shot” Deal

- **V5R3 was the biggest release for CL compiler enhancements since ILE CL compiler in V3R1**
  - Most new CL compiler function since System/38 1.0
- **V5R3 was just the beginning!**
- **More CL compiler enhancements delivered in V5R4, 6.1, 7.1 and 7.2 releases of IBM i**

## Notes:

Well, that's a summary of the set of new and enhanced CL compiler functions and CL documentation enhancements delivered in V5R3. It's interesting to note that this was the biggest release for the CL compiler since the introduction of the ILE CL compiler in V3R1. Also, in terms of new CL compiler function, V5R3 was the biggest release since release 1.0 of System/38 in 1980 when CL was “born”.

But V5R3 was (thankfully) not the end of CL enhancements.

The next set of slides will cover the CL compiler enhancements delivered with V5R4 of IBM i.

And V5R4 didn't mark the end of IBM's investment in CL. Additional incremental enhancements were also added in IBM i 6.1, 7.1, and 7.2 releases.



# Subroutines

- Simple code block between SUBR and ENDSUBR commands
- Invoked by new CALLSUBR command
  - No argument/parameter passing
  - Optional RTNVAL can specify 4-byte \*INT variable
  - No local scoping of subroutine variables
  - Local scoping of labels
  - No nesting allowed (subroutines in subroutines)
- Return to caller via RTNSUBR or ENDSUBR
- Cannot use GOTO to enter or leave the body of a subroutine, OK within a subroutine

## Notes:

We've heard CL programmer's suggestions and complaints that CL needs the ability to define a block of code which can be run from different places in the CL program. This function was delivered in V5R4 through CL subroutines.

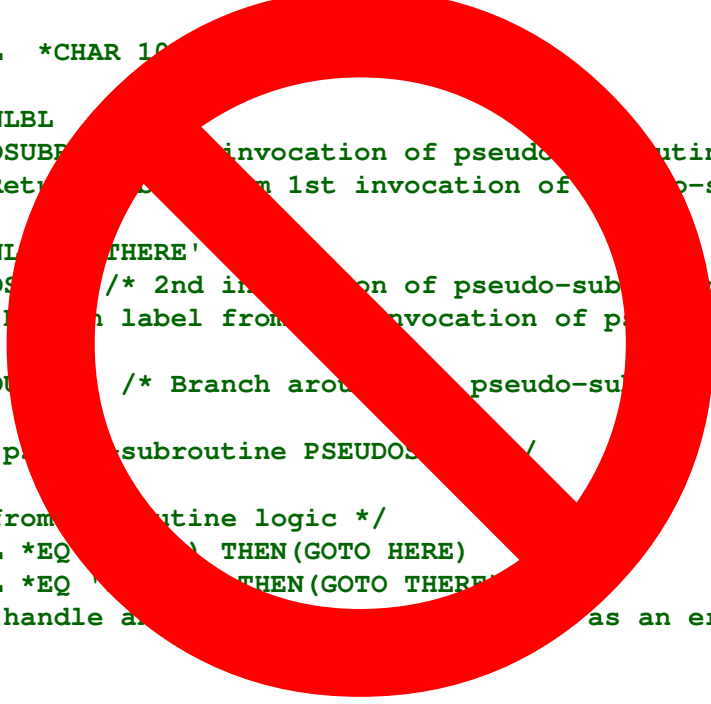
CL subroutines do not have local storage or parameters. All DCL, DCLF, and COPYRIGHT commands must still go at the beginning of the CL program or ILE CL procedure; they are not allowed in a CL subroutine.

CL subroutines can return a 4-byte integer value. This can be used, for example, to indicate whether the operation performed by the subroutine was successful or not.

You cannot nest subroutines; in other words you cannot have a SUBR command within a subroutine. Subroutines can call other subroutines or recursively call the same subroutine by using the CALLSUBR command.

The CL compiler enforces that any GOTO command in a subroutine must reference a label defined within the same subroutine. The mainline CL code (i.e. the code not in any subroutine) cannot reference a label inside a subroutine. CL will allow the same label name to be used in more than one subroutine.

## CL program using a pseudo-subroutine



```

PGM
DCL &RTNLBL *CHAR 10
:
CHGVAR &RTNLBL
GOTO PSEUDOSUBR /* 1st invocation of pseudo-subroutine */
HERE: /* Return label from 1st invocation of pseudo-subroutine */
:
CHGVAR &RTNLBL 'THERE'
GOTO PSEUDOSUBR /* 2nd invocation of pseudo-subroutine */
THERE: /* Return label from 2nd invocation of pseudo-subroutine */
:
GOTO GO_AROUND /* Branch around pseudo-subroutine code */
PSEUDOSUBR:
/* Body of pseudo-subroutine PSEUDOSUBR */
:
/* Return-from-subroutine logic */
IF (&RTNLBL *EQ 'HERE') THEN(GOTO HERE)
IF (&RTNLBL *EQ 'THERE') THEN(GOTO THERE)
/* Need to handle all other return labels as an error */
GO_AROUND:
:
ENDPGM

```

## Notes:

The lack of CL subroutine support has not stopped creative CL programmers from coming up with ways to mimic subroutines ... sort of like how CL programmers had to mimic loops before DOFOR, DOWHILE and DOUNTIL were added in V5R3.

For example, by setting a label name into a CL variable, and branching to a block of pseudo-subroutine code, the value of the CL variable could be tested to determine where to branch back to. Problems with this approach included:

- Any new invocations of the subroutine would require adding logic in the subroutine to handle the new return label
- Have to add GOTO before the subroutine code to avoid “falling in” to the subroutine
- Lots of GOTOs which make the code less structured

## CL program using a real subroutine

```
PGM
:
CALLSUBR REALSUBR /* 1st call to the subroutine */
:
CALLSUBR REALSUBR /* 2nd call to the subroutine */
:
/* End of program mainline code */

SUBR REALSUBR
/* Body of subroutine REALSUBR */
:
RTNSUBR
:
ENDSUBR

ENDPGM
```

## Notes:

With real subroutines, the code is cleaner and more structured.

- Easier to understand the control flow because of explicit CALLSUBR command
- Avoids repetition of common blocks of CL code
- Cannot accidentally fall into the subroutine code
- Return point is always the first command following the CALLSUBR
- Subroutines can call other subroutines and CL keeps track of the subroutine stack.

## Sample CL with Subroutines

```

PGM  /* A simple program with two subroutines */
DCLPRCOPT  SUBRSTACK(25)
DCL  VAR(&SRTNVAR) TYPE(*INT) LEN(4)
DCL  VAR(&SDEC) TYPE(*DEC) LEN(5 2)
MONMSG MSGID(CPF0822) EXEC(GOTO CMDLBL(DUMP))

START:
CALLSUBR  SUBR(SUBR1) RTNVAL(&SRTNVAR)
DUMP:
DMPCLPGM

START:  SUBR          SUBR(SUBR1)
        CALLSUBR     SUBR(SUBR2)
        ENDSUBR      RTNVAL(&SDEC)

START:  SUBR          SUBR(SUBR2)
        CALLSUBR     SUBR(SUBR2)
        RTNSUBR      RTNVAL(-1)
        ENDSUBR

ENDPGM

```

## Notes:

There is an error in this example, because SUBR2 will keep calling SUBR2. CL subroutine recursion is allowed, and it is the CL programmer's responsibility to keep from letting the code go into an infinite subroutine call loop.

The CL runtime will not let the subroutine call stack grow indefinitely. By default, up to 99 entries are allowed on the subroutine stack. Calling a subroutine adds an entry to the stack and returning from the subroutine pops the entry off the stack.

The DCLPRCOPT (Declare Processing Options) command will allow the subroutine stack to be set from 20 to 9999 entries. In this example, the CL runtime will signal CPF0822 when the code attempts to add the 26<sup>th</sup> subroutine stack entry.

The MONMSG command declared at the start of the program will get control when the CPF0822 escape message is signaled and branches to the DMPCLPGM (Dump CL Program) command. Part of the spooled file produced by DMPCLPGM is shown on the next slide.

# DMPCLPGM Output shows Subroutine Stack

```

File . . . . . : QPPGMDMP                               Page/Line  1/6
Control . . . . :                               Columns  1 - 78
Find . . . . . :
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...
                                     Messages
Time      Message      Sev      Type      Message      Fro
164026   CPF0822         40      ESC      Subroutine stack overfl QCL
                                     ow at statement 001600.
                                     Variables
Variable      Type      Length      Value
&SDEC         *DEC      5 2        0
&SRTNVAR     *INT      4          0
                                     Subroutines
Current Stack Depth . . . . . : 25
Current Stack (First 10 entries). . . : SUBR1
                                          SUBR2
                                          SUBR2
                                          SUBR2
F3=Exit  F12=Cancel  F19=Left  F20=Right  F24=More keys
More...

```

## Notes:

Dump CL Program (DMPCLPGM) command is your friend! This is a handy command that every CL programmer should be familiar with.

This slide shows part of the output of DMPCLPGM when the CPF0822 error occurred when the subroutine example program was run.

Note that it shows the exception message information.

Also note the subroutine stack information that is dumped. Only the first 10 subroutine stack entries will be dumped, but if you're in a CALLSUBR loop, that should be enough information.

The dumped information also includes the current values for all CL variables in the program.

# Pointer CL Variables

- Added TYPE(\*PTR) on DCL statement
- New %ADDRESS built-in to **set pointer**
- New %OFFSET built-in to **set or retrieve the offset** portion of pointer variable
- Added STG(\*BASED) attribute on DCL statement
- Enables calling (or being called by) programs that have pointer parameters
- Makes many procedure APIs available to ILE CL
  - Full record-level file I/O
  - String functions

## Notes:

Some people would not expect CL and pointers to be used in the same sentence! This enhancement is part of the effort to make CL more API-friendly. The fact is that many APIs either expect parameters that are pointers or return arrays of information, which are most naturally navigated using pointers and based variables.

We added pointer support in V5R4, with the related support for based variables and built-in functions, %ADDRESS and %OFFSET, to manipulate pointer variables. While both the classic and ILE CL compilers will benefit from the addition of pointer support, the ILE CL compiler may have the advantage because this support will make many API procedures available that the classic CL compiler cannot take advantage of.

# Defined-on CL Variables

- Added STG(\*DEFINED) keyword on DCL command
- Must give name of defined-over CL variable (new DEFVAR keyword on DCL command)
- Can optionally provide starting position (default = 1) from beginning of the defined-over CL variable
- Useful for varying-character fields and providing CL mapping of structures

## Notes:

By allowing a CL variable to be defined over some portion of a larger CL variable, this can be used to provide structure-like support in CL.

For example, you can declare a regular character variable that is the size of a database file record and then declare defined CL variables for each field in the record format.

However, you do not need to declare defined variables for every field. You can simply declare the defined variables for those fields of interest in your program. Later examples will illustrate this capability.

## Examples of V5R4 DCL Enhancements

- **Declaring \*DEFINED CL Variables**

```
DCL &QUALOBJ *CHAR LEN(20)
DCL &OBJ *CHAR LEN(10) STG(*DEFINED) DEFVAR(&QUALOBJ 1)
DCL &LIB *CHAR LEN(10) STG(*DEFINED) DEFVAR(&QUALOBJ 11)
```

- **Declaring a Pointer CL Variable**

```
DCL &CHARVAR *CHAR LEN(10)
DCL &PTRVAR *PTR ADDRESS(&CHARVAR)
```

- **Declaring a \*BASED CL Variable**

```
DCL &ENTRYPTR *PTR
DCL &CHARENTRY *CHAR 10 STG(*BASED) BASPTR(&ENTRYPTR)
```

- **Declaring a \*DEFINED Pointer CL Variable**

```
DCL &CHARSTRUCT *CHAR LEN(48)
DCL &PTRVAR2 *PTR STG(*DEFINED) DEFVAR(&CHARSTRUCT 17)
```

## Notes:

Here are examples that start out pretty simple and get more complex.

The first example shows how you can avoid using %SUBSTRING by declaring the two parts of a 20-character qualified object name as defined. To set the library name can be done by simply changing &LIB rather than sub-stringing the &QUALOBJ variable. Note the new STG parameter and DEFVAR parameter.

The second example shows how to declare a simple pointer variable and initialize it to point to the storage location of the &CHARVAR variable. Note that the initialization is done using the new ADDRESS parameter rather than the VALUE parameter.

The third example shows a declare for a simple based variable. Program working storage will not be allocated for this variable when the CL program or procedure is called; it gives a view of whatever storage location is addressed by its basing pointer. Any variable declared with STG(\*BASED) must specify its basing pointer on the BASPTR parameter of the DCL. When using the based variable, because the basing pointer is known, no pointer qualification is needed.

The last example illustrates how some of these new CL variables can be used together. For example, a pointer might be defined over part of a larger structure. That structure could in turn be based on a pointer, though in this example the structure variable &CHARSTRUCT is declared as a simple variable that is allocated when the CL program or procedure is called.



## Example CL of DCL Enhancements

```

PGM
  DCL &QUALOBJ *CHAR LEN(20)
  DCL &OBJ *CHAR LEN(10) STG(*DEFINED) DEFVAR(&QUALOBJ 1)
  DCL &LIB *CHAR LEN(10) STG(*DEFINED) DEFVAR(&QUALOBJ 11)
  DCL &CHARVAR *CHAR LEN(10)
  DCL &PTRVAR *PTR ADDRESS(&CHARVAR)
  DCL &ENTRYPTR *PTR
  DCL &CHARENTRY *CHAR 10 STG(*BASED) BASPTR(&ENTRYPTR)
  DCL &CHARSTRUCT *CHAR LEN(32) VALUE('FIRSTENTRY')
  DCL &PTRVAR2 *PTR STG(*DEFINED) DEFVAR(&CHARSTRUCT 17)

/* Set values into the CL variables */
CHGVAR &OBJ 'MYMSGF'
CHGVAR &LIB 'MYLIB'
CHGVAR &CHARVAR 'ABCDEFGHJIJ'
CHGVAR &PTRVAR2 (%ADDRESS(&CHARSTRUCT))
CHGVAR &ENTRYPTR &PTRVAR2

/* Dump the program to show values of CL variables */
DMPCLPGM
ENDPGM

```

## Notes:

This shows how references to these new types of CL variables are still pretty simple.

The CHGVAR commands used to set values into the object name and library name variables could have been done using sub-stringing, but this is easier to understand and less code.

Setting a pointer to the address of a local variable can be done at runtime by using the new %ADDRESS built-in function on the VALUE parameter of a CHGVAR command.

Copying a pointer variable to another pointer variable is pretty simple too, accomplished by having pointers for both the VALUE and VAR parameters of a CHGVAR command.

## Using Based & Defined Variables

- **Accessing API output data using %SUBSTRING**

```
DCL &LISTHDR *CHAR 192
IF COND(%SST(&LISTHDR 104 1) = 'C') THEN(DO)
```

- **Accessing API output data using \*DEFINED variable**

```
DCL &LISTHDR *CHAR 192
DCL &LISTSTS *CHAR 1 STG(*DEFINED) DEFVAR(&LISTHDR 104)
IF COND(&LISTSTS = 'C') THEN(DO)
```

- **Accessing API output directly using pointer to user space**

```
DCL &USRSPCTR *PTR
DCL &LISTHDR *CHAR 192 STG(*BASED) BASPTR(&USRSPCTR)
DCL &LISTSTS *CHAR 1 STG(*DEFINED) DEFVAR(&LISTHDR 104)
IF COND(&LISTSTS = 'C') THEN(DO)
```

## Notes:

These code snippets show how using the new \*DEFINED variables is simple and can actually improve the readability of your CL.

The first code snippet is what you might code today if your CL calls one of the system 'List-type' APIs (i.e. one of the APIs shipped with IBM i that returns a list of things, like objects or jobs). At the beginning of the returned data is a header and the one-character field at position 104 of the header is a status field that contains a 'C' if all available entries were returned. Up to V5R4, the way to check this status field was to substring into the header.

In V5R4, you can declare a one-character CL variable which is defined over the header. Referencing the list status field this way is a bit faster but also more obvious since the field being tested has a name rather than just a string of substringed bytes.

Taking this up a notch, in order to handle really big lists, you might want to call the List API and have it store the information in a user space instead of a local \*CHAR variable. In this case, the header would be declared as a \*BASED variable which would have a basing pointer that points to the beginning of the user space. Note that the declare for the list status variable would look the same and the reference to the variable would also be the same.

## Example CL Using Pointer Variable

```

PGM   PARM(&SCANVALUE  &NEWVALUE)

/* Replace all occurrences of parameter &SCANVALUE */
/* in &STRING with parameter &NEWVALUE           */
DCL  VAR(&SCANVALUE) TYPE(*CHAR) LEN(1)
DCL  VAR(&NEWVALUE)  TYPE(*CHAR) LEN(1)
DCL  VAR(&STRING)   TYPE(*CHAR) LEN(26) +
      VALUE('This is the string to scan')

DCL  VAR(&BASEPTR)  TYPE(*PTR) ADDRESS(&STRING)
DCL  VAR(&SCANFOUND) TYPE(*CHAR) LEN(1) +
      STG(*BASED) BASPTR(&BASEPTR)

DCL  VAR(&CONTROLS) TYPE(*CHAR) LEN(8) +
      VALUE(X'010000000000001A') +
/* Start scan with a string length of 26 bytes */
DCL  &RTNCDE TYPE(*INT) LEN(4)

```

## Notes:

This example shows how a pointer variable and based variable can be used in conjunction with one of the many string functions available as callable ILE procedures. The `_SCANX` function could not have been called by CL in prior releases because it requires a pointer parameter.

## Example CL Using Pointer Variable (continued)

```

CHGVAR VAR(%SST(&CONTROLS 4 1)) VALUE(&SCANVALUE)
CALLPRC PRC('_SCANX') PARM((&BASEPTR) +
(&CONTROLS) (X'48000000' *BYVAL)) +
RTNVAL(&RTNCDE) /* Scan the string */

/* Loop to find all occurrences of &SCANVALUE */
DOWHILE COND(&RTNCDE=0) /* Scan value found? */
CHGVAR VAR(&SCANFOUND) VALUE(&NEWVALUE)
CALLPRC PRC('_SCANX') PARM((&BASEPTR) +
(&CONTROLS) (X'48000000' *BYVAL)) +
RTNVAL(&RTNCDE) /* Scan string again */
ENDDO

/* Output the resulting string in a message */
SNDPGMMSG MSG(&STRING) TOPGMQ(*EXT)
ENDPGM

```

## Notes:

The `_SCANX` procedure will scan the string pointed to by `&BASEPTR` looking for the first occurrence of the `&SCANVALUE` character.

If no occurrence is found, a non-zero return code will cause the code to stop replacing occurrences of `&SCANVALUE` with `&NEWVALUE`.

If an occurrence is found, `_SCANX` changes the address of `&BASEPTR` to point to that occurrence. The `CHGVAR` (Change Variable) command, using the based variable `&SCANFOUND`, is used to change that occurrence.

# Debug using Existing Tools

```

Program:  SCANX           Library:  VIG           Module:  SCANX
20      DOWHILE  COND(&RTNCDE=0) /* Scan value found? */
21      CHGVAR  VAR(&SCANFOUND) VALUE(&NEWVALUE)
22      CALLPRC PRC('_SCANX')  PARM((&BASEPTR) (&CONTROLS) +
23      (X'48000000' *BYVAL)) +
24      RTNVAL(&RTNCDE) /* Scan string again */
25      ENDDO
26      /* Output the resulting string in a message */
27      SNDPGMMSG MSG(&STRING) TOPGMQ(*EXT)
28      ENDPGM
  
```

Debug . . . Bottom

F3=End program F6=Add/Clear breakpoint F10=Step F11=Display variable  
 F12=Resume F17=Watch variable F18=Work with watch F24=More keys  
 &STRING = 'This, is, the string to scan'

M& a M& 11/2025  
 I902 - Session successfully started

## Notes:

This slide shows what you would see if the previous example CL was compiled specifying `DBGVIEW(*SRC)` on the `CRTBNDCL` (Create Bound ILE CL Program) command, and you started the source-level Debug function by specifying `STRDBG` and the name of the CL program.

The debug functions work like they do for other language programs that support pointers and based and defined variables.

## Using \*BASED and \*DEFINED with List API

```

/* Generic header information */
DCL      VAR(&USRSPCPTR) TYPE(*PTR)
DCL      VAR(&LISTHDR) TYPE(*CHAR) STG(*BASED) +
        LEN(192) BASPTR(&USRSPCPTR)
DCL      VAR(&LISTSTS) TYPE(*CHAR) STG(*DEFINED) +
        LEN(1) DEFVAR(&LISTHDR 104)
DCL      VAR(&LISTENTOF5) TYPE(*INT) STG(*DEFINED) +
        DEFVAR(&LISTHDR 125)
DCL      VAR(&LISTENTNBR) TYPE(*INT) STG(*DEFINED) +
        DEFVAR(&LISTHDR 133)
DCL      VAR(&LISTENTSIZ) TYPE(*INT) STG(*DEFINED) +
        DEFVAR(&LISTHDR 137)

/* List object detail information */
DCL      VAR(&LISTENTPTR) TYPE(*PTR)
DCL      VAR(&LISTENT) TYPE(*CHAR) STG(*BASED) +
        LEN(30) BASPTR(&LISTENTPTR)
DCL      VAR(&OBJNAME) TYPE(*CHAR) STG(*DEFINED) +
        LEN(10) DEFVAR(&LISTENT 1)
DCL      VAR(&OBJTYPE) TYPE(*CHAR) STG(*DEFINED) +
        LEN(10) DEFVAR(&LISTENT 21)

```

## Using \*BASED and \*DEFINED (continued)

```

/* Miscellaneous variables */
DCL      VAR(&CURRENT) TYPE(*INT)
DCL      VAR(&TEXT) TYPE(*CHAR) LEN(50)

/* Test to see if user space exists */
CHKOBJ   OBJ(QTEMP/OBJLIST) OBJTYPE(*USRSPC) MBR(*NONE)

/* Create user space if not found */
MONMSG   MSGID(CPF9801) EXEC(CALL PGM(QUSCRTUS) +
        PARM('OBJLIST QTEMP ' 'QUSLOBJ ' +
        X'0000001' X'00' '*CHANGE ' 'Output from +
        QUSLOBJ API'))

/* Get list of all objects in the QTEMP library */
CALL     PGM(QUSLOBJ) PARM('OBJLIST QTEMP ' +
        'OBJL0100' '*ALL QTEMP ' '*ALL')

/* Get pointer to user space */
CALL     PGM(QUSPTRUS) PARM('OBJLIST QTEMP ' +
        &USRSPCPTR)

```

## Using \*BASED and \*DEFINED (continued)

```

/* Was the list built successfully? */
IF COND((&LISTSTS = 'C') *AND (&LISTENTNBR > 0)) +
  THEN (DO)
/* Address the first list entry */
  CHGVAR      VAR(&LISTENTPTR) VALUE(&USRSPCPTR)
  CHGVAR      VAR(%OFFSET(&LISTENTPTR)) +
              VALUE(%OFFSET(&LISTENTPTR) + &LISTENTOFS)

/* Process all returned entries */
  DOFOR      VAR(&CURRENT) FROM(1) TO(&LISTENTNBR)
/* Write the object information */
  CHGVAR      VAR(&TEXT) VALUE('Object: ' +
                              *CAT &OBJNAME +
                              *CAT ' Type: ' *CAT &OBJTYPE)
  SNDPGMMSG  MSG(&TEXT) TOPGMQ(*EXT)
/* Address the next list entry */
  CHGVAR      VAR(%OFFSET(&LISTENTPTR)) +
              VALUE(%OFFSET(&LISTENTPTR) + &LISTENTSIZ)

ENDDO

  SNDPGMMSG  MSG('End of list') TOPGMQ(*EXT)
ENDDO

```

## Notes:

This example pulls together several of the concepts and code snippets introduced earlier. The program will check if it the output user space and create it if the CHKOBJ command signals a CPF9801 (Object not found) escape message.

The QUSLOBJ (List Objects) API is called and the output is directed to the user space. API QUSPTRUS is called to get a pointer to the data returned. The first thing in the returned data is a header which indicates if all available entries were returned, the offset to the first entry, the number of entries, and a bunch of other information that this program isn't interested in. Notice how the CL variables declared as \*DEFINED in the header only declare the parts of the header which are of interest to the program.

The %OFFSET built-in function is used to set the basing pointer for the list entry structure. Here again, only the "interesting" fields in the entry structure are declared as \*DEFINED variables. Once the list entry has been processed, the %OFFSET function is used to bump the list entry basing pointer to the next entry.

This is done in a DOFOR loop that has a terminating value of the number of entries which is a \*DEFINED variable in the list header.

# More File I/O in CL

## DDS for record format for database file FILEA

```
R RECORD
  KEYVALUE      5  0
  DATA         10
  MOREDATA     10
K KEYVALUE
```

## CL that defines the same record format for FILEA

```
DCL &RECORD  *CHAR LEN(23)           /* Record buffer */
DCL &RECORDSIZE TYPE(*INT) VALUE(23) /* Record length */
DCL &KEY *DEC LEN(5 0) STG(*DEFINED) DEFVAR(&RECORD)
DCL &KEYSIZE TYPE(*INT) VALUE(3) /* Size of key (bytes) */
DCL &DATA *CHAR LEN(10) STG(*DEFINED) DEFVAR(&RECORD 4)
DCL &MOREDATA *CHAR LEN(10) STG(*DEFINED) DEFVAR(&RECORD 14)
```

## Notes:

Input/output support in CL has always been very limited. For database files, CL could only read a file sequentially ... no writing to database, no keyed I/O, no record updates, deletes, etc. This is changing in V5R4 because CL can now use the rich set of record I/O procedures shipped with the operating system and used by other languages, like C and C++.

The next several slides show CL code doing record-level I/O to a simple keyed physical file named FILEA. The record format for FILEA contains just three fields.

One disadvantage of doing I/O using the record I/O APIs is that you need to write the CL declares manually that map to the file record format structure.



## CL for opening database file FILEA

```

DCL &NULL TYPE(*CHAR) LEN(1) VALUE(X'00')
DCL &FILENAME *CHAR 11 /* File to be worked with */
DCL &MODE *CHAR 50 /* Open modes */
DCL &RFILE *PTR /* File pointer */

/* File feedback information */
DCL &FEEDBACK TYPE(*PTR) /* I/O feedback pointer */
DCL &FDBCKDATA TYPE(*CHAR) STG(*BASED) +
    LEN(64) BASPTR(&FEEDBACK) /* Feedback data */
DCL &NUM_BYTES TYPE(*INT) STG(*DEFINED) +
    DEFVAR(&FDBCKDATA 37) /* Number of bytes processed */

/* Open FILEA by calling _Ropen which returns file pointer */
CHGVAR VAR(&FILENAME) VALUE('FILEA' *TCAT &NULL)
CHGVAR VAR(&MODE) VALUE('rr+' *TCAT &NULL)
CALLPRC PRC('_Ropen') PARM((&FILENAME) (&MODE)) +
    RTNVAL(&RFILE)

```

## Notes:

Because these record-level I/O APIs were written for ILE C in V2R3, input strings (like the name of the file to open and the mode to open the file) need to be null-terminated strings.

Because the `_Ropen` API required a pointer parameter (`&RFILE` in this example), ILE CL could not call it in releases before V5R4.

## CL for reading a record by key

```

PGM   PARM(&KEYIN &DATAIN)
      :
      DCL  &KEYIN   *DEC LEN(15 5)  /* Key of record for update   */
      DCL  &DATAIN  *CHAR LEN(10)   /* Value to update record with */
      DCL  &KEY     *DEC LEN(5 0) STG(*DEFINED) DEFVAR(&RECORD)
      DCL  &OPTS    TYPE(*INT)      /* Options for _Rreadk       */
      DCL  &KEY_EQ  TYPE(*INT) VALUE(X'0B000100') /* Equal to Key             */
      DCL  &RTNCDE  TYPE(*INT)      /* Return code                */
      :
      CHGVAR VAR(&KEY) VALUE(&KEYIN)
      CHGVAR VAR(&OPTS) VALUE(&KEY_EQ)

/* Call _Rreadk to try to read record with key field = &KEYIN */
CALLPRC PRC('_Rreadk') +
        PARM((&RFILE *BYVAL) (&RECORD) (&RECORDSIZE *BYVAL) +
              (&OPTS *BYVAL) (&KEY) (&KEYSIZE *BYVAL)) +
        RTNVAL(&FEEDBACK)

```

## Notes:

Notice that the `_Rreadk` (Read Record by Key) API requires input parameters of the file pointer returned from `_Ropen`. It is defined that the parameter be passed “by value” which ensures that `_Rreadk` cannot change the pointer.

The feedback structure returned by `_Rreadk` will indicate whether a record was found with the specified key. A value of zero for ‘Number of bytes processed’ indicates no record was found with the specified key.

## CL for adding or updating a record

```

/* If no record has matching key, write a new record */
IF COND(&NUM_BYTES *EQ 0) THEN(DO)
  CHGVAR VAR(&DATA) VALUE(&DATAIN)
  CHGVAR VAR(&MOREDATA) VALUE(' ')
  CALLPRC PRC('_Rwrite') +
    PARM((&RFILE *BYVAL) (&RECORD) (&RECORDSIZE *BYVAL)) +
    RTNVAL(&FEEDBACK)
ENDDO

/* Record was read successfully, update the existing record */
ELSE (DO)
  CHGVAR VAR(&DATA) VALUE(&DATAIN)
  CALLPRC PRC('_Rupdate') +
    PARM((&RFILE *BYVAL) (&RECORD) (&RECORDSIZE *BYVAL)) +
    RTNVAL(&FEEDBACK)
ENDDO

```

## Notes:

This CL code tests the feedback data returned from `_Rreadk` and will either write a new record or replace the existing record.

- If no record was found in the file that had a matching key field, a new record is written to the file by calling the `_Rwrite` (Write Record) API.
- If a record was found with the specified key, the existing record is replaced in the file by calling the `_Rupdate` (Update Record) API.

# Reading the Contents of a Directory

(by Scott Klement)

PGM

```
DCL VAR(&NUL)      TYPE(*CHAR) LEN(1)  VALUE(X'00')
DCL VAR(&DIRNAME) TYPE(*CHAR) LEN(200)
DCL VAR(&EXT)      TYPE(*CHAR) LEN(4)
DCL VAR(&POS)      TYPE(*INT)  LEN(4)
DCL VAR(&SUCCESS) TYPE(*INT)  LEN(4)
DCL VAR(&HANDLE)   TYPE(*PTR)
DCL VAR(&ENTRY)    TYPE(*PTR)
DCL VAR(&NULLPTR) TYPE(*PTR)
DCL VAR(&STMF)     TYPE(*CHAR) LEN(640)
```

```
/******  
/* Directory entry returned from readdir. It's 796 chars */  
/* long with a field called &NAMELEN in positions 53-56 */  
/* and a field called &NAME in positions 57-696 */  
/******  
DCL VAR(&DIRENT)  TYPE(*CHAR) LEN(796) +  
DCL VAR(&NAMELEN) STG(*DEFINED) DEFVAR(&DIRENT 53) +  
                  TYPE(*UINT) LEN(4)  
DCL VAR(&NAME)    STG(*DEFINED) DEFVAR(&DIRENT 57) +  
                  TYPE(*CHAR) LEN(640)
```

## Notes:

This example is used by permission of the author, Scott Klement. When Scott learned that CL would be adding support for pointers, he wasn't that interested initially. Later he thought about some of the powerful sets of Integrated File System (IFS) APIs that he was calling from ILE RPG routines he had written and he reconsidered the value of pointers in CL.

This CL will open a directory and process the files in the directory looking for ones with a particular file extension. The data in those files will be merged into a single data base file and then an RPG program (which is not shown) is called to process the information in the database file.

## Reading the Contents of a Directory (continued)

```

/*****
/* Open the /edi/incoming/custdata directory: */
/* the API returns a NULL pointer to indicate */
/* failure. */
/*****
CHGVAR VAR(&DIRNAME) VALUE('/edi/incoming/tabdata' *CAT &NUL)
CALLPRC PRC('opendir') PARM((&DIRNAME)) RTNVAL(&HANDLE)
IF (&HANDLE *EQ &NULLPTR) DO
    SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGDTA('Unable +
        to open directory!') MSGTYPE(*ESCAPE)
ENDDO

/*****
/* Read the contents of the directory */
/*****
CALLPRC PRC('readdir') PARM((&HANDLE *BYVAL)) RTNVAL(&ENTRY)

```

## Notes:

Like the record I/O APIs, the input directory name passed to the opendir (Open Directory) API must be a null-terminated string.

Note how the code is checking whether the directory was opened successfully. If the opendir is not successful, the parameter pointer variable &HANDLE will be set to null. However, CL doesn't have a \*NULL special value that could be used on the COND of the IF command. Instead, the code checks the &HANDLE pointer against a local pointer value that was not set to any value, &NULLPTR. This enables &HANDLE to be compared to another pointer which has a null value. Not quite as pretty as it could be, but it works!

If the opendir APIs returns a valid pointer, it is passed as a "by value" (i.e. input only) parameter to the readdir (Read Directory) API.

## Reading the Contents of a Directory (continued)

```

/* Copy each stream file that ends with .TAB to PF TRANMAST */
DOWHILE COND(&ENTRY *NE &NULLPTR)
  CHGVAR VAR(&STMF) VALUE(%SST(&NAME 1 &NAMELEN))

/* extract the last 4 characters from the filename */
IF (&NAMELEN *GE 4) DO
  CHGVAR VAR(&POS) VALUE(&NAMELEN - 3)
  CHGVAR VAR(&EXT) VALUE(%SST(&STMF &POS 4))
ENDDO
ELSE +
  CHGVAR VAR(&EXT) VALUE(' ')

/* if stream file extender is '.TAB', copy to physical file + and
delete stream file (so it doesn't get processed again) */
IF ((&EXT *EQ '.TAB') *OR (&EXT *EQ '.tab')) DO
  CPYFRMIMPF FROMSTMF(&STMF) TOFILE(TRANMAST) MBROPT(*ADD) +
    RCDDL(*CRLF) FLDDL(*TAB) RPLNULLVAL(*FLDDFT)
  DEL OBJLNK(&STMF)
ENDDO
CALLPRC PRC('readdir') PARM((&HANDLE *BYVAL)) RTNVAL(&ENTRY)
ENDDO

```

## Notes:

The DOWHILE loop will keep calling the readdir API until all directory entries (i.e. files in the directory) have been read.

Inside the loop, the \*DEFINED variables which map over the directory entry returned by readdir are used to check the name of the file. If it is one of the “interesting” table files, the CL runs the CPYFRMIMPF (Copy From Import File) to copy the data to a database file. The table file data is appended so that the database file will contain the information from all the table files found in the directory.

Once the table file has been copied, it is deleted by using the DEL (Delete Object) command.

## Reading the Contents of a Directory (continued)

```

/*****/
/* Close the directory */
/*****/

CALLPRC PRC('closedir') PARM(&HANDLE *BYVAL) +
      RTNVAL(&SUCCESS)

/*****/
/* Call an RPG program to process the new */
/* transactions in the physical file */
/*****/

CALL PGM(PROCESS)

ENDPGM

```

## Notes:

After all the directory entries have been processed, the `closedir` (Close Directory) API is called to unlink the directory.

Scott then calls an ILE RPG program to process all the data in the physical file. It's worth noting that RPG is a much more powerful programming language than CL will ever be and is a better choice than CL for applications that require lots of business logic.

# Dynamic Storage using ILE C Library Functions

```

/*****/
/* Do CRTCLMOD followed by CRTPGM with BNDDIR(QC2LE) */
/*****/
DCL &RCVVARPTR TYPE(*PTR)
DCL &RCVVARsiz TYPE(*INT) VALUE(256)
DCL &RCVVAR *CHAR 256 STG(*BASED) BASPTR(&RCVVARPTR)
DCL &BYTRTN *INT 4 STG(*DEFINED) DEFVAR(&RCVVAR)
DCL &BYTAVL *INT 4 STG(*DEFINED) DEFVAR(&RCVVAR 5)
DCL &CRTDATE *CHAR 13 STG(*DEFINED) DEFVAR(&RCVVAR 65)

CALLPRC PRC('malloc') PARM((&RCVVARsiz *BYVAL) +
RTNVAL(&RCVVARPTR)

CALL PGM(QUSROBJD) PARM(&RCVVAR &RCVVARsiz +
'OBJD0100' 'QGPL QSYS ' +
'*LIB ')

SNDUSRMSG MSG(&CRTDATE) MSGTYPE(*INFO) TOMSGQ(*EXT)

CALLPRC PRC('free') PARM((&RCVVARPTR *BYVAL))

```

## Notes:

This example shows how CL can use the new pointer variables and based variable support in conjunction with ILE C dynamic storage library functions.

These C runtime library functions are in service programs shipped with IBM i. In order to find these API procedures, the CL developer would need to create the CL program in two steps:

- 1) Run the Create CL Module (CRTCLMOD) command to compile the CL source.
- 2) Run the Create Program (CRTPGM) command to bind the \*MODULE with the service programs exported by binding directory QC2LE to create the ILE \*PGM object

In this example, the size to be allocated is essentially a constant, but you can see how the program could easily be modified to allocate a variable number of bytes. The malloc() function could be used to allocate up to 16 million bytes of dynamic storage.

If 16MB is not enough, there are teraspace storage library functions similar to malloc() and free() that can allocate and free huge chunks of virtually contiguous storage.

The malloc() function reserves a block of storage of the specified size. This is commonly referred to as *heap storage*.

The free() function frees up the storage allocated using malloc(). That chunk of heap storage could then be reallocated by a subsequent malloc() or calloc() function.



# Summary (so far)

- Enhancements in V5R3 made CL a better application development language and let you do more in your CL programs
- Further CL compiler enhancements in V5R4 really opened up what a CL can do
- You've seen just a few examples that demonstrate some of the ways the CL compiler enhancements can extend what you can do in a CL program or ILE CL procedure

## Notes:

OK, so now we've covered enhancements to the Control Language in both V5R3 and V5R4 releases of the operating system.

These enhancements have made it easier to write more structured programs in CL by using standard HLL constructs like DO loops and SELECT statements. Integers are a regular data type. CL variables can be defined over other CL variables to make it easy to reference individual fields in structures. Pointers and based variables, along with new %OFFSET and %ADDRESS built-in functions enable CL programs to pass pointers to other programs or use API programs and API procedures that require pointers or pass-by-value parameters.

CL programs and ILE CL procedures can now invoke any API program or API procedure, which greatly enhances CL for writing simple application programs.

Now we will cover the next set of enhancements to the CL language that IBM delivered in IBM i 6.1.

# V6R1 -- CL Compiler

- New **CLOSE** command
  - Ability to read a database file multiple times
  - Next RCVF after a CLOSE implicitly reopens the file
  - Can use with OVRDBF to read multiple file members or reposition to a different record before next open
- New **INCLUDE** command
  - Ability to imbed additional CL source at compile time
  - Similar to **/COPY** in RPG or **#include** in C/C++
  - INCFILE parameter added to CRTCLxxx commands
  - New option (RTVINCSRC) added to RTVCLSRC

## Notes:

The CLOSE command allows a file declared in a CL program or CL procedure (using DCLF) to be explicitly closed. CL *implicitly* closes files opened using DCLF when the program or procedure ends. The problem was that once you tried using RCVF to read past the end of the file and got the “end of file” (CPF0864) error, it was “game over” for trying to use the file. Starting in V6R1, you can use the CLOSE command to explicitly close the file and the next RCVF command implicitly reopens the file. Used in conjunction with the OVRDBF (Override Database File), you can use CLOSE to read all the members of a database file by changing the MBR parameter. You could also change the POSITION parameter to start with a record other than the first record in the member when the file is opened by the first RCVF command run after the CLOSE.

The INCLUDE (Include CL Source) command allows source to be imbedded in a CL source program during compilation. This allows you to share common source between multiple CL programs, which can reduce CL application maintenance costs. The CRTCLPGM, CRTCLMOD, and CRTBNDCL commands were be changed to support a new INCFILE (INCLUDE file) parameter which defaults to \*SRCFILE but can be used to specify a common file used for shared CL code. The RTVCLSRC command was changed to support a new RTVINCSRC (Retrieve included source) which will allow the user to specify whether the retrieved source should have the original INCLUDE commands or the CL source from the included source member.

# V6R1 -- CL Compiler (continued)

- Support creation options in CL source code
  - Added many CRTCLxxx parameters to DCLPRCOPT
  - Also BNDSRVPGM & BNDDIR parameters
  - Similar to some types of H-specs in RPG
- Support for \*NULL special value **(ILE CL mostly)**
  - Can specify for ADDRESS value on DCL for TYPE(\*PTR)
  - Can assign to \*PTR variable on CHGVAR command
  - Can specify in pointer expressions (e.g. on IF or WHEN)

## Notes:

Enhanced DCLPRCOPT command to allow in-source specification of object creation parameters normally specified on the CRTCLPGM, CRTCLMOD, or CRTBNDCL command. This makes it simpler and less error prone to recompile CL source code. In addition, DCLPRCOPT will support the BNDSRVPGM (Binding service program) and BNDDIR (Binding directory) parameters normally specified on the CRTPGM (Create Program) command, which will enable a CL source program compiled using the CRTBNDCL command to create an ILE (Integrated Language Environment) CL program that automatically binds with the desired \*SRVPGM and \*MODULE objects.

Support new \*NULL special value for setting or testing CL pointer variables when compiling CL source code using CRTCLMOD or CRTBNDCL. For example, ADDRESS(\*NULL) can be specified on the DCL for a CL pointer variable, or COND(&STRPTR \*EQ \*NULL) could be specified on the IF command.

For CRTCLPGM (i.e. original CL), \*NULL is allowed on the DCL for a \*PTR CL variable, but not in a comparison or assignment statement (e.g. COND parameter or CHGVAR command).

# V7R1 -- CL Compiler

- Support 8-byte integer CL variables (**ILE CL only**)
  - Allow LEN(8) for TYPE(\*INT) or TYPE(\*UINT) on DCL command
  - Passing parameters between CL and other HLL programs
    - For example, ILE RPG and C/C++ support 8-byte integer variables
  - Handle 8-byte integers returned by IBM i APIs
    - For example, disk I/O and CPU amounts returned by QUSRJOB API
- Optional DO/SELECT level indicator in compile listings
  - New \*DOSLTLVL value for OPTION parameter on CRTCLxxx
- Nested INCLUDE support
  - No compiler limit on level of nesting (watch out for recursion)
  - Also available on 6.1 via PTF

## Notes:

The Declare CL Variable (DCL) command supports a value of 8 for the LEN parameter for signed integer (\*INT) and unsigned integer (\*UINT) variables if the CL source is compiled using the CRTCLMOD command or the CRTBNDCL command. This capability is useful when calling API programs and API procedures that define 8-byte integer fields in input or output structures.

You can specify OPTION(\*DOSLTLVL) on the Create CL Program (CRTCLPGM) command or the CRTCLMOD command or the CRTBNDCL command. This compiler option adds a new column to the compiler listing which shows the nesting levels for Do (DO), Do For (DOFOR), Do Until (DOUNTIL), Do While (DOWHILE), and Select (SELECT) commands.

The INCLUDE command support that was shipped in 6.1 was enhanced to allow an INCLUDE command in the CL source brought in by an INCLUDE command. Be careful because you could get recursion if included member X contains an INCLUDE command for member Y and that CL contains an INCLUDE command for member X. This support was added via PTF SI35660 for IBM i 6.1 and shipped with release 7.1.

# V7R1+ (new function via PTF)

- Decision to tie next release (after 7.1) to Power 8
- Wanted to “pull” IBM i customers to latest release
- Challenge of delivering new function without any new messages, panels, or online help (i.e. MRI PTFs)
- Decided to extend CL built-in function support
  - 1<sup>st</sup> PTF included %TRIM, %TRIML, %TRIMR
  - 2<sup>nd</sup> PTF included %CHECK, %CHECKR, %SCAN

## Notes:

The 7.1 release of IBM i was fairly closely associated with Power 7 hardware. The “powers that be” decided that the next release of IBM i would be delivered in concert with Power 8 hardware. Nice idea, but as the hardware dates moved out, so did the operating system date.

IBM i started delivering new support, particularly for attached devices, to Technical Refreshes (or TRs) of the Licensed Internal Code (or LIC). New operating system function (especially DB2/SQL) would line up with the LIC TR dates. The idea was to show continual improvement to IBM i which would incent i customers to upgrade to the current release (i.e. 7.1).

The tricky part for delivering new function mid-release to operating system components like the CL compiler is avoiding changes that would require new translatable information such as new messages or new panels. In IBM-speak, this means avoiding MRI PTFs which are **very** time-consuming to build, test, and approve.

Jennifer Liu (technical owner of the CL compiler) took the challenge and delivered new CL built-in functions that extended existing CL BIF support (e.g. %SST). This support was added via PTFs SE53451 and SI49061 for IBM i 7.1 in early 2013.

## New 7.1 Built-ins %TRIM, %TRIML, %TRIMR

- Trim characters from variable
  - %TRIM(char-variable-name [chars-to-trim])
  - %TRIMR(char-variable-name [chars-to-trim])
  - %TRIML(char-variable-name [char-to-trim])
  - Required parameter: Type(\*Char) variable to be trimmed
  - Optional parameter: Type(\*Char) variable or literal string of characters to be trimmed. Default is the blank character
  - **Can be used on V5R4 and 6.1 by specifying TGTRLS on CRT**



Tip

```
Dcl      Var(&Amount) Type(*Char) Len(15) +
          Value('*****12.50***')
```

```
Dcl      Var(&Result) Type(*Dec)
```

```
ChgVar   Var(&Result) Value(%TRIM(&Amount ' *'))
```

&Result is now 12.50

*i want an i.*

© 2014 Guy Vig

## New 7.1 Built-ins %CHECK and %CHECKR

- Check for (or verify) characters within a variable
  - %CHECK(chars-to-check char-variable-name [start-position])
  - %CHECKR(chars-to-check char-variable-name [start-position])
  - Required parameter 1: Type(\*Char) variable or literal string of characters to check for
  - Required parameter 2: Type(\*Char) variable to be checked
  - Optional parameter: Type(\*Int), Type(\*Uint) or Type(\*Dec) variable or literal numeric value with zero decimal positions. Default is 1 for %CHECK, the size of char-variable-name for %CHECKR
  - Returns 0 if all characters in char-variable-name in chars-to-check, otherwise location within char-variable-name of character not in chars-to-check. Return value can be Type(\*Int), Type(\*Uint), or Type(\*Dec)
  - **Can be used on V5R4 and 6.1 by specifying TGTRLS on CRT**



Tip

```
Dcl      Var(&PhoneNbr) Type(*Char) Len(20) +
          Value(' (507) 993-022X')
```

```
If      Cond(%CHECK(' -()0123456789' &PhoneNbr) *NE 0) +
          Then(SndPgmMsg Msg('Invalid telephone number'))
```

Returned value would be 14 (i.e the relative location of 'X' within &PhoneNbr)

*i want an i.*

© 2014 Guy Vig

## New 7.1 Built-in %SCAN

- Scan for a pattern within a variable
  - **%SCAN(search-argument source-string [start-position])**
  - Required parameter: Search-argument can be Type(\*Char) variable or literal string of pattern to scan for
  - Required parameter: Source-string can be Type(\*Char) variable to be scanned or the special value \*LDA
  - Optional parameter: Start-position can be Type(\*Int), Type(\*Uint) or Type(\*Dec) variable or literal numeric value with zero decimal positions. Default is 1
  - Returns 0 if search-argument not found in source-string, otherwise location within source-string where start of search-argument found. Return value can be Type(\*Int), Type(\*Uint), or Type(\*Dec)
  - **Can be used on V5R4 and 6.1 by specifying TGTRLS on CRT**



Tip

```
Dcl Var(&Text) Type(*Char) Len(50) +
  Value('CL now supports %CHECK and %SCAN')
Dcl Var(&Pos) Type(*Int)
```

```
ChgVar Var(&Pos) Value(%Scan('%S' &Text))
```

**&Pos value would be 28 (i.e. the starting relative location of 'S' within &Text)**

## Notes:

The CL compilers have historically had only a handful of built-in functions and limited string manipulation capabilities. You could reference a subset of a character CL variable by using the %SUBSTRING (or %SST) built-in function and do character string concatenation by using \*CAT, \*TCAT, and \*BCAT operators (or ||, |<, <|).

This support was improved significantly by the addition of six more built-in functions that were delivered via PTFs to 7.1 IBM i.

More examples can be found in the IBM Knowledge Center article on [Built-in Functions for CL](#).

Due to the amount of change to the CL compilers, this support was not PTF'ed to earlier releases. However, since the changes only affected the generated code (i.e. no changes required to CL runtime functions), you can compile CL or CLLE code on a 7.1 system with the PTFs installed and specify TGTRLS(V6R1M0) or TGTRLS(V5R4M0), save the generated CL objects for V5R4 or 6.1, and restore the CL objects on a V5R4 or 6.1 system and run them.

# V7R2 delivered more CL BIFs

- Some new CL BIFs required changes to CL runtime
  - Can't compile with TGTRLS of V6R1M0/V7R1M0 & use these BIFs
- My retirement cut the size of the “CL team” by half
- New CL built-in functions delivered in three areas:
  - Type conversion (%CHAR, %DEC, %INT, %UINT)
  - String case conversion (%UPPER, %LOWER)
  - Space (%LEN, %SIZE)

## Notes:

Delivering new CL built-in functions without any new messages or commands was a pretty neat trick, but required that the function be done completely in the generated code. For the type conversion built-in functions, for example, the CL compiler design does some of the work by calling CL runtime programs shipped as part of IBM i. This made it impractical (never say impossible) to allow using these BIFs and still compiling the CL or CLLE source with a TGTRLS of V6R1M0 or V7R1M0.

Though I was only working part-time on CL when I retired in 2013, I was handling a good portion of the maintenance work, primarily providing Level 3 service for customer-reported questions and problems. This allowed Jennifer to focus the majority of her time on new function.

The new CL built-in functions delivered as part of the 7.2 release of IBM i fall into three categories. The first group allows the CL developer to convert a string to a numeric value or vice-versa. The second group handles case conversion of a string from lower case to upper case or vice-versa and, unlike the %XLATE built-in function in RPG, is done correctly based on CCSID. The third set lets you determine either the declared length of a CL variable or the number of bytes occupied by a CL variable.



## New 7.2 Built-in %CHAR

- Convert a CL variable value to a character string
  - **%CHAR(convert-argument)**
  - Required parameter: Convert-argument must be CL variable with TYPE of \*LGL, \*DEC, \*INT or \*UINT
  - Leading zeroes will be suppressed in the resulting character string
  - Decimal point character dependent on QDECFMT system value
  - %CHAR can be used anywhere that CL supports a character expression
  - **Can only be used in CL created and run on a 7.2 system**

```
Dcl Var(&Temp) Type(*Int) Value(-10)
Dcl Var(&Msg) Type(*Char) Len(60)
```

```
ChgVar Var(&Msg) Value('Current temperature is' +
  *BCAT %CHAR(&Temp) *BCAT 'degrees Centigrade')
```

**&Msg** would have the value **'Current temperature is -10 degrees Centigrade'**

## New 7.2 Built-in %DEC

- Convert CL variable to a decimal numeric value
  - **%DEC(convert-argument [total-digits decimal-places])**
  - Required parameter: Convert-argument must be CL variable with TYPE of \*CHAR, \*LGL, \*DEC, \*INT or \*UINT
  - Optional parameters allow control of total digits and decimal digits in result
  - If input variable is \*CHAR, sign is optional and can precede or follow, decimal point is optional and can be period or comma, leading and trailing blanks are allowed, and an all-blank value returns a value of zero.
  - %DEC can be used anywhere that CL supports an arithmetic expression
  - **Can only be used in CL created and run on a 7.2 system**

```
Dcl Var(&Points) Type(*Char) Len(10) Value('-123.45')
Dcl Var(&Answer) Type(*Dec) Len(10 5)
```

```
ChgVar Var(&Answer) Value(100 + %DEC(&Points 5 2))
```

**&Answer** would have the value **-00023.45000**

## New 7.2 Built-ins %INT and %UINT

- Convert CL variable to an integer numeric value
  - **%INT(convert-argument) or %UINT(convert-argument)**
  - Required parameter: Convert-argument must be CL variable with TYPE of \*CHAR, \*LGL, \*DEC, \*INT (for %UNIT) or \*UINT (for %INT)
  - If input variable is \*CHAR, sign is optional and can precede or follow, decimal point is optional and can be period or comma, leading and trailing blanks are allowed, and an all-blank value returns a value of zero.
  - Decimal digits are truncated without rounding and the result is a 4-byte integer.
  - %INT and %UNIT can be used anywhere CL supports an arithmetic expression
  - %UNS allowed for compatibility with RPG and will be handled same as %UINT
  - **Can only be used in CL created and run on a 7.2 system**

```
Dcl  Var(&Points) Type(*Char) Len(10) Value('-123.45')
Dcl  Var(&Answer) Type(*Int)
```

```
ChgVar Var(&Answer) Value(100 + %INT(&Points 5 2))
```

**&Answer** would have the value -23

## New 7.2 Built-ins %LOWER and %UPPER

- Convert string to all upper case or all lower case
  - **%LOWER(input-string [CCSID]) and %UPPER(input-string [CCSID])**
  - Required parameter: Input-string must be CL variable with TYPE of \*CHAR
  - Optional parameter: CCSID of input string, default is to use the job CCSID
  - Functionally equivalent to UPPER and LOWER built-in functions in SQL
    - Correctly handles characters based on coded character set ID (CCSID)
  - **Can be used on 6.1 and 7.1 by specifying TGTRLS on CRT**

```
Dcl  Var(&Str) Type(*Char) Len(12) /* 'Hello World!' */ +
     Value(X'48656C6C6F20576F726C6421') /* in CCSID 819 */
```

```
ChgVar Var(&Str) Value(%LOWER(&Str))
```

**&Str** would have value **'hello world!'** in CCSID 819 (ISO/ANSI Multilingual)

## New 7.2 Built-ins %LEN and %SIZE

- Return number of digits/characters or bytes for CL variable
  - **%LEN(variable-argument)** and **%SIZE(variable-argument)**
  - Required parameter: Variable-argument must be CL variable.
    - For %LEN, the CL variable TYPE must be \*CHAR, \*DEC, \*INT or \*UINT
    - For %SIZE, the CL variable TYPE can be any valid type, including \*PTR
  - %LEN and %SIZE can be used anywhere CL supports an arithmetic expression
  - %LEN used with character or decimal CL variables returns the declared length
  - %LEN used with integer CL variables returns the maximum possible digits
  - %SIZE always returns the number of bytes occupied by the CL variable
  - **Can be used on 6.1 and 7.1 by specifying TGTRLS on CRT**

```
Dcl  Var(&Rtn) Type(*Int) Len(2)
Dcl  Var(&Num1) Type(*Dec) Len(7 2)
ChgVar Var(&Rtn) Value(10 + %LEN(&Num1))
&Rtn would have the value 17 (i.e. 10 + 7)
ChgVar Var(&Rtn) Value(10 + %SIZE(&Num1))
&Rtn would have the value 14 (i.e. 10 + 4)
```

**A couple “future”  
CL compiler  
enhancements  
under evaluation**

## Possible fix for CALL passing constant values

- **Allow specifying length of constant parameter values**
  - Current behavior is confusing (at least at first) for those new to CL
    - Character constants passed as 32 bytes or length of constant literal value
    - Numeric constants passed as packed decimal with 15 digits and 5 decimal digits
  - More confusion when CALL imbedded in SBMJOB with CL variable parms
    - Command string needs to be a literal string in order to be passed to new job
    - Any CL variables passed as parameters get converted to character constants
  - Changing a CALL to SBMJOB with embedded CALL appears to "break"
    - Parameter storage often appears to be "garbage"
  - New second element for each PARM value to specify parameter length

```
Call Pgm(Lib1/MyPgm) Parm(('ABC' (50)) (1234 (12 2))
```

**1st parameter would be passed as \*Char 50, 2nd parameter as \*Dec 12 2**

```
SbmJob Cmd(Call Pgm(Pay/UpdPay) Parm(&LastName (50)) +
  (&FirstName (40)) (&Salary (12 2))
```

**Pass &LastName as \*Char 50, &FirstName as \*Char 40, &Salary as \*Dec 12 2**

## Change CALL/CALLPRC to allow expressions

- Both commands already allow literals for parameters
  - **CALL PGM(QCMDEXC) PARM(strtcp 6)**
  - **CALLPRC PRC(PAYROLL) PARM(CHICAGO 1234 &VAR1)**
- Neither command allows an expression (currently)
  - Can't use concatenation, substring, or any of the new CL built-in functions
- New support would allow expressions for program/procedure parms
  - Like specifying expression for a CL command parameter defined with EXPR(\*YES)
  - CALL from non-compiled CL would still not allow expression (obviously?)
- Depending on hits to CL compiler and runtime, may be 7.2 PTF only

# Thank YOU

# Questions

